Erweiterung eines metamodell-basierten Generator-Frameworks zur Erzeugung von Quellcode für eine mehrschichtige Client/Server Architektur am Beispiel von J2EE

Fachhochschule Stuttgart
Hochschule der Medien
Diplomarbeit im Studiengang Medieninformatik

Vorgelegt bei
1. Prüfer: Herrn Prof. Dr. Edmund Ihler, Hochschule der Medien, D-70569 Stuttgart
2. Prüfer: Herrn Dipl.-Ing. Klaus Banzer, Softlab GmbH, D-81677 München

Christian Märkle

München, 28. Februar 2004

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst, und keine anderen als die aufgeführten Hilfsmittel in Anspruch genommen habe. Ich habe ausschließlich die angegebenen Quellen verwendet und mich auch sonst keiner nichterlaubten Hilfsmittel bedient.

Darüber hinaus bestätige ich, die Arbeit bisher weder im In- noch Ausland als Prüfungsarbeit eingereicht zu haben.

Datum

Unterschrift

DANKSAGUNG

Mein Dank gilt zunächst dem Unternehmen Softlab GmbH, das es auch in wirtschaftlich schlechten Zeiten für wertvoll erachtet, Diplomarbeiten anzubieten und an Studenten zu vergeben.
Dort gilt mein Dank insbesondere Michele Siegler, der die Einstellung möglich gemacht hat und mich auch in meiner weiteren beruflichen Laufbahn unterstützt hat. Vielen Dank an meinen Betreuer Klaus Banzer, der mich in kompetenter Weise beraten und mit konstruktiver Kritik weitergebracht hat. Als weitere Kollegen bei Softlab möchte ich Markus Reiter, Klaus Kubo und Maximilian Klauser erwähnen, die gleich zu Beginn der Diplomarbeit für eine sympathische Atmosphäre und die nötige „Wohlfühl-Temperatur" gesorgt haben.
An der Hochschule der Medien möchte ich den Dank an Prof. Dr. Edmund Ihler richten, der die Diplomarbeit von Seiten der Hochschule betreut und durch die Vorlesung Informatik 4 den nötigen Unterbau für die Diplomarbeit geliefert hat. Weiterhin ist Prof. Walter Kriha zu nennen, der das Forum zum Thema MDA möglich gemacht hat und dessen Vorlesungen absolute Praxisrelevanz haben.
Und zuletzt herzlichen Dank an meinen Kommilitonen Markus Wichmann, mit dem ich mich aus studentischer Sicht über die Diplomarbeit austauschen konnte und der mir auch layouttechnisch weitergeholfen hat.

SOFTLAB GMBH

Das Unternehmen mit Hauptsitz in München ist eine 100% BMW-Group-Company und entwickelt unter anderem kundenspezifische Individual-Lösungen auf Basis neuer Technologien. Im Vordergrund steht die Betreuung von Kunden aus Industrie, Automotive, Finanzdienstleister und Handel während des gesamten Prozesses von der Konzeption über die Integration bis zur Unterstützung im laufenden Betrieb.

Anschrift

Softlab GmbH
Zamdorfer Straße 120
D-81677 München
www.softlab.de

ZUSAMMENFASSUNG

Die technische Infrastruktur in Client/Server Architekturen großer verteilter Anwendungen basiert oftmals auf mehreren Schichten mit immer wiederkehrenden Mustern, als Beispiel sei hier im Besonderen die Komponentenarchitektur von J2EE zu nennen. Die Implementierung dieser Infrastruktur nimmt viel Zeit in Anspruch und steht in den meisten Fällen in keinem Verhältnis zu ihrer intellektuellen Herausforderung an die Entwickler. Hier liegt es nahe, solcherlei Aufgaben durch einen generativen Ansatz zu lösen.

Von vielen Ansätzen hat in den letzten Jahren der von der Object Management Group (OMG) spezifizierte Model Driven Architecture (MDA) Ansatz ein großes Interesse in der Fachwelt hervorgerufen, der über die reine Generierung von Code hinausgeht und gar eine Revolution in der Software-Entwicklung auslösen soll.

Die Diplomarbeit beschäftigt sich zunächst mit den Spezifikationen, die MDA zugrunde liegen und führt dann über die Basistechnologien wie J2EE und der BMW Component Architecture zum Einsatz eines MDA konformen Generator-Frameworks, mit dessen Hilfe der praktische Einsatz getestet werden soll. Die Diplomarbeit beschreibt die Anpassung und die Möglichkeiten des Frameworks und setzt sie anhand zweier praktischer Beispiele aktueller Software-Projekte der Softlab GmbH im Bereich Supply Chain Management (SCM) ein.

Ein Rückblick zum Ende der Diplomarbeit setzt sich mit den aufgetretenen Problematiken und Fallstricken, aber auch den Chancen von MDA kritisch auseinander und bewertet deren praktischen Nutzen. Der anschließende Ausblick gibt die persönliche Einschätzung des Autors über die zukünftige Entwicklung von MDA wieder und stellt einige weiterführende Ideen für Diplomarbeits-Themen vor.

HINWEISE ZUM LESEN DER DIPLOMARBEIT

Die Ausführung der Diplomarbeit bestand aus einem theoretischen und praktischen Teil. Dies spiegelt sich auch in der Inhaltsstruktur des Berichts wieder.

Zunächst führen Kapitel 1 und 2 in die zugrundeliegende Aufgabenstellung und den Lösungsansatz ein.

Kapitel 3 beschäftigt sich mit der theoretischen Basis und der Spezifikation zu MDA und den dort verwendeten Konstrukten wie UML, MOF und XMI. Dieses Kapitel orientiert sich stark an den von der Object Management Group erhältlichen Dokumenten.

In Kapitel 4 wird die Architektur und das Application Programming Interface (API) der Zielplattform J2EE beschrieben, mit dem Schwerpunkt auf der Enterprise JavaBeans Technologie. Die für das Anwendungsdesign vorgebene Rahmenarchitektur wird in Kapitel 5 näher erläutert, während sich Kapitel 6 der Funktionsweise des für den Transformationsprozess eingesetzten Generator Frameworks widmet.

Mit Kapitel 7 als umfangreichstem Kapitel kommt der praktische Teil der Diplomarbeit zum Ausdruck. Dort werden die implementierten Generator-Templates, die Erweiterung des Metamodells sowie die allgemeine Konfiguration und das Deployment der generierten Anwendungen beschrieben.

Kapitel 8 zeigt die praktische Anwendung der in Kapitel 7 beschriebenen Erweiterungen anhand zweier Projektebeispiele.

Kapitel 9 und 10 schließen die Diplomarbeit mit einem zusammenfassenden Rückblick und einer kritischen Bewertung der zukünftigen Entwicklung von MDA ab.

Ein Auflistung der in der Diplomarbeit verwendeten Literatur und sonstiger Ressourcen ist in den Anhängen zu finden.

INHALTSVERZEICHNIS

1 Einleitung

In den letzten Jahren ist von verschiedenen Software-Projekten in der IT-Industrie oder in der Wissenschaft vermehrt zu hören, dass Code-Generatoren eingesetzt wurden, um nach bestimmten Vorgaben und Mustern Programm-Code nicht mehr von Hand durch den Entwickler, sondern von einem speziellen Programm erzeugen zu lassen.

Im Grunde genommen ist dieser Ansatz jedoch nichts Neues, denn in den 50er und 60er Jahren des 20. Jahrhunderts wurde bereits Code generiert. Damals wurden Abfolgen von Maschinenbefehlen zu Funktionen zusammengefasst und in Anweisungen einer Programmiersprache mit höherem Abstraktionsgrad abgebildet. Eine Sammlung von Funktionen diente als Bibliothek für den Anwendungsprogrammierer, der diese einfach verwenden konnte, ohne über ihre Umsetzung (in Maschinenbefehlen) Bescheid zu wissen. Ein Compiler interpretierte und generierte aus den Anweisungen wiederum die ursprünglichen Maschinenbefehle, mit der die Rechenmaschine angesteuert werden konnte.

Hier wird auch schon ein Vorteil deutlich: Immer wiederkehrende und logisch zusammengehörende Abfolgen von Anweisungen werden abstrahiert, d. h. für den Entwickler unwichtige Details werden versteckt. Der Entwickler nutzt ausschließlich die abstrakte Notation, um Programme zu entwickeln. Ein C-Anwendungsprogrammierer muss sich nicht mehr dafür interessieren, wie die Ausgabe eines Textes mit `printf()` in Maschinenbefehlen umgesetzt wird. Dadurch spart er Zeit, Geld, schreibt fehlerfreie Programme und kann sich auf das Wesentliche konzentrieren, nämlich die Lösung des an ihn gestellten Problems. Natürlich ist Code-Generierung nicht zu verwechseln mit sogenannten generischen Komponenten wie Bibliotheksfunktionen und Frameworks, die in der selben Sprache verfasst sind wie die programmierte Anwendung. Eine Bibliotheksfunktion befindet sich auf der selben Ebene der Abstraktion, da sie ausschließlich Befehle logisch gruppiert.

Wird nun der damalige Ansatz konsequent weiterverfolgt, so muss man heutzutage eine weitere Ebene der Abstraktion hinzufügen, um den konkreten Anforderungen bei der Umsetzung von verteilten Software-Systemen gerecht zu werden. Eine Umsetzung davon ist bereits im Einsatz von virtuellen Maschinen zu sehen, die es erlauben, Programmcode zu entwickeln, der von dem eingesetzten Betriebssystem der darunterliegenden Plattform weitgehend unabhängig ist. Die Java Virtual Machine (JVM) etwa interpretiert plattformunabhängigen Bytecode und ruft plattformabhängige Betriebssystemfunktionen auf.

Ein weiterer Schritt nach vorne ist die grafische Modellierung von Software-Systemen, aus denen in mehreren Verfeinerungsschritten konkreter und ausführbarer Programmcode erzeugt werden kann.

Die Object Management Group (OMG), eine Organisation, deren Mitglieder sich aus der IT-Industrie, Wissenschaft und Hochschulen zusammensetzen, hat sich in den letzten Jahren zum Ziel gesetzt, einen Standard zu entwickeln, der den Entwurf und die Modellierung von Software-Systemen von technischen Abhängigkeiten und Details abstrahiert und damit u. a. die Kommunikation zwischen Kunden und Entwicklern vereinfachen soll.
Die sogenannte Model Driven Architecture (MDA) nutzt bereits vorhandene Spezifikationen wie etwa die Unified Modelling Language (UML), um Modelle grafisch oder textuell zu stan-

dardisieren und dadurch die Entwicklung von Generatoren und Transformatoren zu vereinfachen, die Modelle schlussendlich in ausführbaren Programmcode überführen sollen.

Die Diplomarbeit beschäftigt sich mit dem Konzept der modell-getriebenen Architektur, gibt deren Vor- und Nachteile wieder und zeigt dessen pragmatische Anwendung anhand der Generierung einer J2EE Architektur aus einem technisch unabhängigen Modell, das in UML notiert wurde.

2 Aufgabenstellung

2.1 Ausgangsituation

Als Grundlage der Diplomarbeit diente zunächst das aktuelle Projekt *International Vehicle System – Reengineering (IVS-R)*, das ein neues web-basiertes Online Bestellsystem für die BMW Gruppe zur Verfügung stellt. Das System wird von BMW Händlern dazu benutzt, Fahrzeuge für ihre Kunden bei der BMW AG zu bestellen. Alle Bestellungen werden online ausgeführt und erhalten eine direkte Bestätigung über die Machbarkeit und das Produktionsdatum aus der verantwortlichen Fabrik. Das System wurde zunächst im amerikanischen Markt und im nahen Osten eingeführt, da in diesen Märkten bisher keine Online-Bestellmöglichkeiten existierten. Als nächster Schritt ist die Ablösung von Altsystemen (IVS – International Vehicle System) im deutschen und europäischen Markt geplant.

Das Projekt wird im Rahmen einer internationalen Kooperation durchgeführt. Softlab arbeitete bei Analyse/Design und Implementierung der Backend Kernfunktionalität (Business Logik) eng mit der Fachabteilung des BMW Vertriebes zusammen. Das Frontend wurde von einem Team der Vertriebsgesellschaft BMW North America implementiert. Organisatorisch ist das Projekt im Bereich Supply Chain Management (SCM) angesiedelt.

IVS-R liegt die BMW Core Server Architecture (CSA) zu Grunde, einer abstrakten Architektur für eine Klasse von Client/Server Software-Architekturen mit gemeinsamen Konzepten und Merkmalen. Die Anwendung wurde für die Java 2 Plattform Enterprise Edition (J2EE) realisiert, das Backend mit Enterprise Java Bean Komponenten, die auf einem BEA Weblogic Application Server laufen.

Eine Teilfunktionalität von IVS-R ist die Pflege der Stammdaten von BMW Händlern, die auf etwa zehn bis fünfzehn Datenbanktabellen verteilt sind. Die relativ simplen und wiederkehrenden Anwendungsfälle wie das Anlegen, Lesen und Löschen von Stammdaten und die Anzahl der Datenbanktabellen sprachen für den Einsatz eines Code-Generators, um die EJB Komponenten zu erzeugen. Der bisher verwendete Generator besteht aus einer Reihe von XML Dateien, die aus der Datenbankstruktur generiert und über XSL Templates in Quellcode transformiert werden. Der Generierungsprozess erfolgte über ein Ant Skript.
Der Generator nutzt also eine „Bottom-Up" Herangehensweise, da der Input die Struktur der Datenbank ist und nicht etwa ein Domänenmodell. Das ist ein sehr praxisorientierter Ansatz, da Änderungen oft zuerst auf der Datenbank gemacht werden und anschließend der Quellcode nachgezogen wird (und wenn man Glück hat, auch die System-Spezifikation).

2.2 Ziele

Das primäre Ziel der Diplomarbeit bestand nun darin, den bisher eingesetzten Mechanismus für die Generierung der EJB Architektur des Backends in ein Framework zu fassen und soweit zu verallgemeinern, dass er auch von Anwendungen anderer Projekte einfacher wiederverwendet werden kann. Es stand zu Beginn der Diplomarbeit noch nicht fest, welcher Ansatz für das Framework und die Abstraktion als Grundlage dienen sollte.
Nach einiger Recherche in Fachzeitschriften und durch die Erfahrung des Model Driven Architecture (MDA) Workshops an der Hochschule der Medien habe ich mich nach Rückspra-

che mit meinem Betreuer dazu entschieden, den Ansatz der von der Object Management Group spezifizierten Model Driven Architecture zu verwenden.
MDA liefert für die Erfüllung des oben genannten Ziels eine fundierte Architektur, die es erlaubt, unabhängig von der generierten Zielplattform ein Anwendungs-Design zu entwickeln, so dass ein wiederverwendbarer Einsatz über Projektgrenzen hinweg möglich ist.
Erfreulicherweise hat sich herausgestellt, das es bereits Implementierungen des MDA Konzepts in Form von Generator Frameworks gibt, so dass sich erheblich Zeit für den Aufwand der Erstellung eines Frameworks einsparen ließ. Der eigentliche Implementierungsaufwand bestand nun darin, ein vorhandenes Framework so zu erweitern und an die vorgegebene BMW Component Architecture (CA) anzupassen, dass eine sinnvolle Generierung von Quellcode für das Backend verwirklicht werden konnte.

Aus der Zielvorgabe und MDA als Paradigma ergaben sich für die Diplomarbeit folgende Anforderungen:

- plattformunabhängige und visuelle Modellierung und Dokumentation des Backends einer verteilten Anwendung in UML
- Refactoring und Anpassung der älteren Component Server Architecture (CSA) an die neue BMW Component Architecture (CA) unter Verwendung bekannter EJB Design Patterns
- generative Erzeugung eines Implementierungsrahmens
- Test und Bewertung der Durchführbarkeit des MDA Ansatzes
- Wiederverwendbarkeit in anderen Projekten

3 Model Driven Architecture (MDA)

MDA ist zur Zeit in aller Munde, die einschlägigen Fachmagazine haben mindestens einen Artikel zu diesem Thema veröffentlicht und auch auf der OOP 2004, Europas grösster Messe rund um das objektorientierte Programmieren, setzen sich viele Vorträge und Workshops damit auseinander.

Model Driven Architecture (MDA) ist ein seit dem Jahr 2001 von der Object Management Group (OMG) entwickelter Standard für die Anwendung und den Einsatz von Modellen in der Software-Entwicklung.

Nach [OMG03] definiert MDA eine Architektur von Modellen, die Richtlinien zur Verfügung stellt, um in Modellen ausgedrückte Spezifikationen zu strukturieren. Oder etwas einfacher ausgedrückt: MDA stellt Richtlinien zur Verfügung, wie verschiedene Modelle strukturiert werden sollten.

Die OMG greift dabei auf bereits vorhandene Standards wie Unified Modelling Language (UML, [OMG01]), Meta Object Facility (MOF, [OMG02]) und XML Metadata Interchange (XMI, [OMG05]) zurück. MDA kann man als konsequente Weiterentwicklung der grafischen Objektmodellierung sehen, die bislang hauptsächlich zu dokumentarischen Zwecken eingesetzt wurde, in manchen Fällen wurde daraus auch bereits technischer Code generiert. MDA führt eine neue Ebene der Abstraktion in die Software-Entwicklung ein und trennt die Spezifikation der Systemfunktionalität von der Spezifikation der plattformspezifischen Implementierung [MDA03].

3.1 Übersicht/Konzept

Modelle

Ein zentraler Bestandteil der MDA ist die Verwendung und der Einsatz von Modellen. Ein Modell beschreibt ein System – wie es strukturiert ist, wie es sich verhält und wie es funktioniert. Die Beschreibung kann in textueller oder in grafischer Form vorhanden sein, üblicherweise tritt sie in Kombination auf. Eine Beschreibung ist dann formalisiert, wenn sie in einer Sprache verfasst ist, die eine festgelegte Form (Syntax) und einen Bedeutungsgehalt (Semantik) hat. MDA stellt mit der Meta Object Facility (MOF) einen Satz von in UML definierten Modellspezifikationen zur Verfügung, auf die man bei der Modellierung zurückgreifen kann. Modelle, die informal beschrieben und nicht MOF konform sind, sind keine gültigen MDA-Modelle. Es genügt also nicht, ein Modell eines Systems mit irgendwelchen Kästchen und Linien zu zeichnen, wenn deren Bedeutung nicht eindeutig nach MOF spezifiziert ist.

Abstraktion

Ein weiterer Schlüsselbegriff ist die Abstraktion. Abstraktion in Modellen versteckt unnötige Details eines Systems, die zum Zeitpunkt des Entwurfs nicht relevant sind. Daraus folgt, dass es zu einer Abstraktion auch eine Realisierung gibt, die dann die Details zur Ausführung der Systemfunktionen enthält. Man kann sich leicht vorstellen, dass es mehrere Stufen der Abstraktion (Levels of Abstraction) geben kann, je nach Blickpunkt auf ein System. So hat ein Projektleiter einer Fachabteilung eine andere Sicht auf ein Buchungssystem als der Systemarchitekt oder der Applikationsprogrammierer. Der Projektleiter möchte die Anwendungsfälle und fachlichen Objekte seiner Domäne sehen, die ihm das System zur Verfügung stellt, während den Anwendungsprogrammierer vor allem die technischen Objekte und Variablen interessieren. MDA sagt nun, dass zwischen den verschiedenen Modellen, die eine Ebene der

Abstraktion abbilden, Verfeinerungen stattfinden, es gibt also eine Verfeinerungsbeziehung zwischen den Modellen. Für die Praxis bedeutet das, dass man zunächst festlegt, aus welchen Blickwinkeln ein System zu betrachten ist und dementsprechend Modelle entwirft. In weiteren Schritten fügt man in einem oder mehreren Transformationsprozessen plattformspezifische Details hinzu, bis man zum tatsächlichen ausführbaren Programmcode kommt.

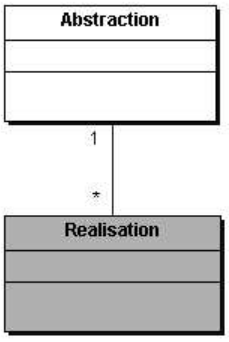

Abbildung 1: Die Beziehung zwischen Abstraktion und Realisation.

Zoom In/Zoom Out

MDA erlaubt bzw. befürwortet die Verknüpfung von Modellen (Ebenen der Abstraktion, Sichten und Blickwinkel) untereinander, was zu interessanten Möglichkeiten führt, die man am besten mit dem Vorgang der Zoomfunktion in einem Fotobearbeitungsprogram wie z. B. Photoshop vergleichen kann. Wenn man sich einen Bildbereich genauer anschauen („hineinzoomen") möchte, wählt man ihn mit einem Selektionswerkzeug aus und erhält eine Vergrößerung, auf der weitere Details sichtbar werden. Analog dazu können Objekte innerhalb eines Modells mit weiteren verfeinerten Modellen verknüpft sein. Wie im Bildbearbeitungsprogramm wird ein Objekt selektiert und man erhält eine komplexe Sicht auf ein Netzwerk mit Objekten zurück (z. B. Stubs, Skeletons, Binders), die durch das Objekt zuvor abstrahiert wurden. Umgekehrt kann man „herauszoomen", um wieder zum vereinfachten Modell zu gelangen. Ankerpunkte für den Zoomvorgang können alle Modellierungselemente innerhalb der Modelle sein.

Dieses Konzept kann ein wichtiger Bestandteil von Entwicklungswerkzeugen sein, die MDA unterstützen. Leider ist dem Autor bisher noch kein Werkzeug bekannt, das dieses Konzept brauchbar umsetzt. Man muss jedoch anmerken, dass eine Implementierung von Zoom-Funktionalität nicht trivial ist, da Design-Änderungen durch den Benutzer in einem Modell von anderen Modellen in den nächsthöheren oder nächstniedrigeren Abstraktionsebenen interpretiert werden müssen, um „Round-Tripping" oder Rücktransformationen zu ermöglichen.

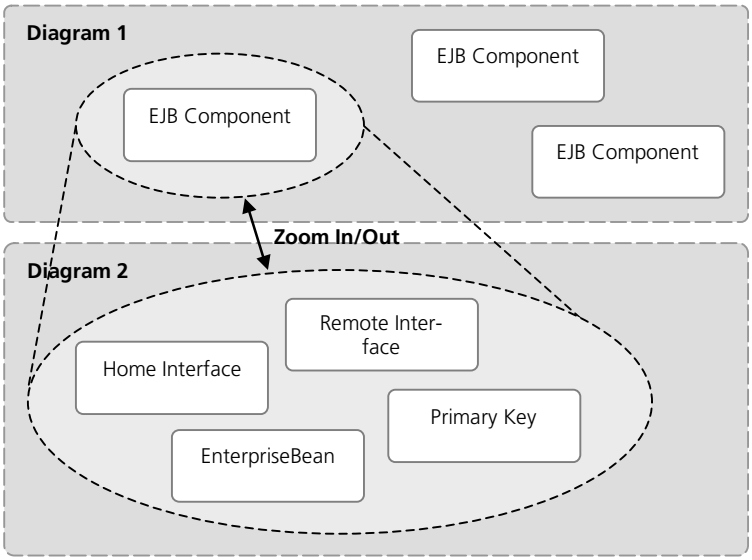

Abbildung 2: Der Zoomvorgang

Plattform

Unter Plattform fasst MDA technische Details zusammen, die irrelevent für die grundlegende Ausführung von Funktionen eines Systems sind [MDA03]. Eine Plattform bietet über Schnittstellen Technologien an, die ein Subsystem verwenden kann, ohne deren genaue Implementierung zu kennen. So nutzt ein Anwendungsentwickler die Win32 API von Microsoft, um Programme für die MS Windows Plattform zu schreiben, ohne dessen genaue Implementierung kennen zu müssen. Als weitere Plattformen lassen sich beispielsweise auch CORBA oder J2EE anführen.

3.2 Modelle: PIM, PSM

MDA definiert zwei wichtige Modelle zur Beschreibung eines Systems. Das Platform Independend Model (PIM) gibt die Sicht auf ein System wieder, die unabhängig von der verwendeten Plattform ist. Dies soll die Anwendung des Modells auf eine Gruppe von ähnlichen Plattformen ermöglichen. So ist eine virtuelle Maschine, die auf einer bestimmten Klasse von Betriebssystemen einsetzbar ist, ein gutes Beispiel für ein PIM. Ebenso ist ein solches Modell vorstellbar für die Beschreibung einer mehrschichtigen Client/Server Architektur, unabhängig davon, ob diese dann in .NET oder J2EE realisiert wird.

Ein Platform Specific Model (PSM) beschreibt dagegen ein System aus der Sicht einer für die Realisierung verwendeten Plattform. Es befindet sich damit auf einer niedrigeren Abstraktionsebene als das PIM. So ist ein PSM zum Beispiel die konkrete Implementierung der Java Virtual Machine für die Linux Plattform, besitzt also eine für Linux kompilierte Laufzeitumgebung. Für MS Windows Betriebsysteme sieht diese dann wieder etwas anders aus. Analoges gilt für das Modell einer Client/Server Architektur, das beispielsweise die .NET Umgebung beschreibt.

Wie oben bereits angeklungen ist, kann ein PIM von mehreren PSMs abstrahieren. Oder anders ausgedrückt: Zwischen beiden Modellen besteht eine 1:n Beziehung.

Ein PIM kann durch einen in Kapitel 3.6 beschriebenen Transformationsprozess in ein oder mehrere PSMs überführt werden. Daraus ergibt sich, dass von mehreren Zielplattformen, für die Modelle entworfen werden, Gemeinsamkeiten vorhanden sein müssen, ansonsten ist es nicht möglich, in geeignetem Maße davon zu abstrahieren. Oder anders ausgedrückt: Zielplattformen müssen einer bestimmten Klasse von Plattformen angehören.

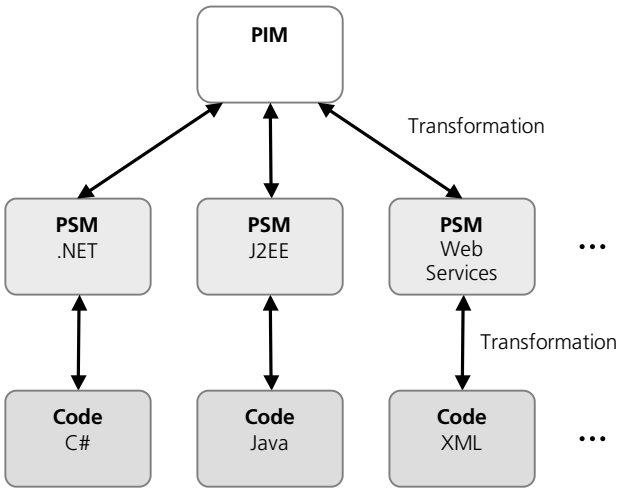

Abbildung 3: Die Modellebenen gemäß MDA

3.3 Metamodelle und MOF

Die Unified Modelling Language [OMG01] wird in der MDA dazu verwendet, PIMs und PSMs zu notieren, sei es nun grafisch (über ein entsprechendes Grafik-Werkzeug) oder textuell (z. B. über XML).

Die Syntax und Semantik von UML wird durch ein sogenanntes Metamodell festgelegt, das selbst in UML notiert ist (vgl. Metamodell-Diagramm in [OMG01]). Ein Metamodell ist vergleichbar mit der Document Type Definition (DTD), die ein XML Dokument beschreibt. Das Metamodell legt die Struktur und die Abhängigkeiten von Modellierungselementen fest, die in einem Modell vorhanden sein dürfen. Es definiert einen abstrakten Kern an „Modellierungselementen", der anschließend durch weitere Metamodelle ausgebaut werden kann.

Mit UML ist es möglich, verschiedene Arten von Diagrammen zu notieren. Es gibt statische Strukturdiagramme (wie etwa das Klassendiagramm), Interaktionsdiagramme, Komponentendiagramme etc. Jedes dieser Diagramme wird durch Erweiterungen des Kern-Metamodells eindeutig beschrieben.

Da Metamodelle selbst ja auch irgendwo definiert und beschrieben sein müssen (sie können nicht im luftleeren Raum hängen), gibt es die sogenannte Meta Object Facility (MOF, [OMG02]), die zum einen eine abstrakte Sprache definiert, mit der Metamodelle und ihre Abhängigkeiten untereinander beschrieben werden können und zum anderen ein Repository an Modelldefinitionen zur Verfügung stellt. Es handelt sich dabei um sogenannte Meta-Metamodelle. Man spricht auch von einer mehrschichtigen Metamodell Architektur zur Spezifikation von Metainformationen, bei der eine Schicht die jeweils darunterliegende beschreibt.

- das MOF Meta-Metamodell beschreibt das UML-Metamodell
- das UML-Metamodell beschreibt UML-Modelle
- ein UML-Modell beschreibt Eigenschaften eines Systems
-

Metamodelle bieten für den Transformationsprozess in MDA den entscheidenden Vorteil, dass sie als Navigationsmöglichkeit durch ein Modell verwendet werden können. Ähnlich des Reflection-Mechanismus´ in Java lassen sich Informationen über die Struktur des Modells zur Laufzeit abfragen, um dann die Transformation in jeweils plattformspezifischen Quellcode zu beeinflussen. Dies ist möglich, wenn ein Metamodell als Implementierung in einer Programmiersprache vorliegt und im Hauptspeicher ausgeführt werden kann. Wie man später sehen wird, spielt dieser Aspekt eine wichtige Rolle in den MDA Werkzeugen.

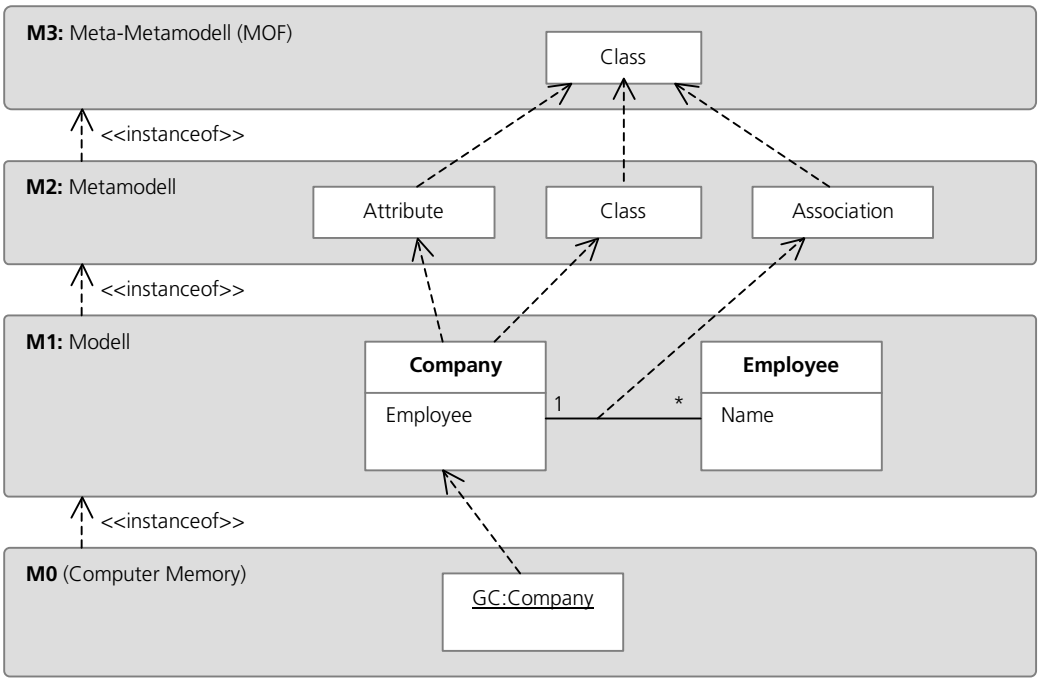

Abbildung 4: Die Metamodell-Schichten

3.4 UML Profile

Profile [OMG04] sind der Erweiterungsmechanismus von UML. UML Profile erlauben es, die Sprache auf eine bestimmte Plattform (z. B. Middleware) oder ein bestimmtes Gebiet der EDV (Domäne) anzupassen. So legt das *UML Profile for Enterprise Distributed Object Computing* (EDOC) fest, wie Standard-UML-Modelle für Verteilte Systeme erweitert werden können, oder das *UML Profile for Schedulability, Performance, and Time* für die Modellierung von

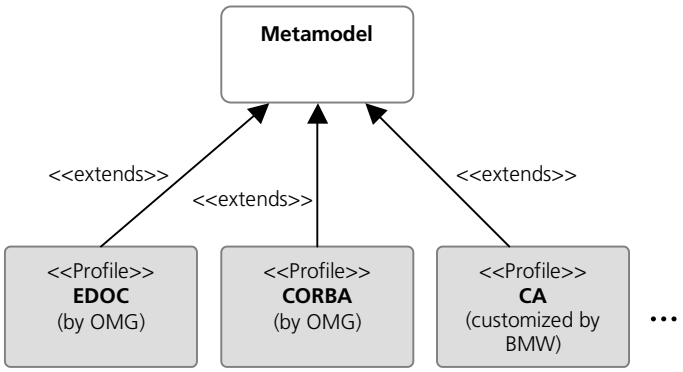

Abbildung 5: UML-Profile erweitern das Metamodell.

Echtzeitsystemen. Zur Zeit sind von der OMG Profile für vier Domänen spezifiziert, weitere sollen noch folgen. Das Profil hilft bei der Modellierung, indem es beispielsweise Stereotypen, Markierungen (Tagged Values) und prüfbare Einschränkungen (Constraints) vorschlägt. Es gibt dem Modell damit die domänenspezifische Semantik, die für den Transformationsprozess von Modellen in der MDA notwendig ist. Die Definition eines neuen Profils wird in der Regel über eine Erweiterung der Klassen des Metamodells ausgedrückt.

3.5 XML Metadata Interchange (XMI)

XMI legt fest, wie Modelle zwischen verschiedenen Modellierungswerkzeugen ausgetauscht werden können [OMG05]. XMI verwendet einen XML-Dialekt für die Beschreibung von Diagrammen und erlaubt damit auch die (De-) Serialisierung von Modellen, um sie persistent zu machen oder über Datennetze übertragen zu können. XMI kommt damit die Rolle des Bindeglieds zwischen UML, MOF und Middleware (UML Profile) zu.

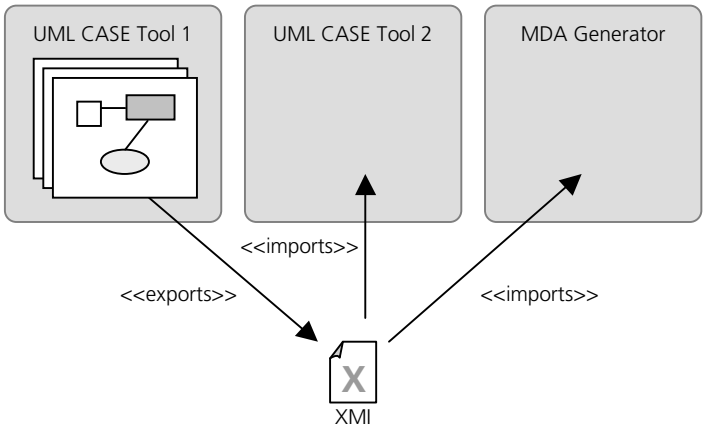

Abbildung 6: XMI als Austauschformat

3.6 Mapping

Die Überführung eines PIM in ein PSM muss durch gewisse Regeln und Techniken festgelegt sein, ansonsten fehlt die Eindeutigkeit oder überhaupt die sinnvolle Ausführbarkeit. Mapping ist daher ein wichiges Merkmal von MDA. Nach [OMG03] gibt es mehrere Varianten, wie ein Mapping ausgeführt werden kann:

- *durch Typprüfung von Metamodellen*: Metamodelle erlauben es, während der Transformation durch das PIM zu navigieren. Dabei können Typprüfungen an Modellierungselementen des Metamodells vorgenommen werden, um das Ergebnis zu beeinflussen.
- *durch Markierung von Modellierungselementen*: Bestimmte Modellierungselemente innerhalb eines PIMs können durch Markierungen (Flags) ausgezeichnet werden, um darauf bei der Transformation zuzugreifen. Die Anreicherung mit Information ist dabei transparent, so dass bei der Transformation immer diejenige Information herausgefiltert wird, die für die jeweilige Plattform wichtig ist
- *durch Templates*: Ein Template erlaubt das Überführen von bestimmten Entwurfsmustern eines PIMs in Entwurfsmuster eines PSM. *Beispiel*: Bei der Transformation eines PIMs wird ein „Entity“ Template verwendet, das ein mit „Entity“ markiertes Objekt in EntityBean, Local und Remote Home Interfaces eines PSMs für die J2EE Plattform überführt.
-

In der Praxis werden oftmals verschiedene Mapping-Varianten miteinander vermischt – PIMs werden mit Markierungen angereichert und in Templates zusammen mit Typprüfungen des Metamodells überprüft.

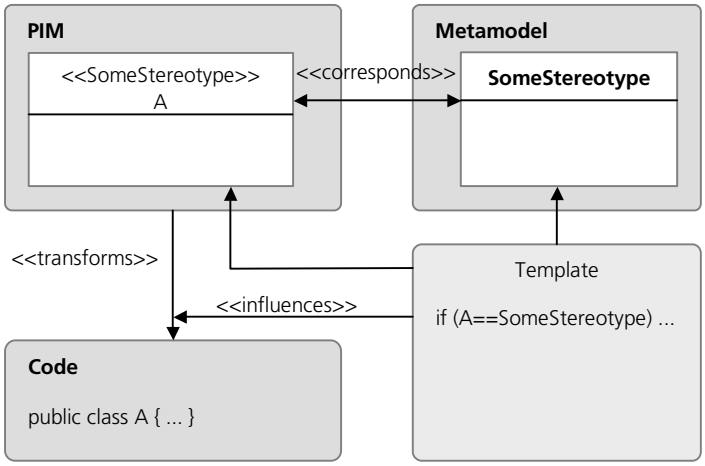

Abbildung 7: Metamodell und Templates beeinflussen den Transformationsprozess

Transformation

Der Transformationsprozess verwendet ein markiertes Modell und ein gewähltes Mapping als Eingabe und wird entweder manuell, teilweise automatisiert oder völlig automatisch ausgeführt. Das Ergebnis einer Transformation ist ein neues Modell oder direkt Quellcode. Zusätzlich kann der Prozess auch aufgezeichnet werden, um das (nicht-) erfolgreiche Mapping zu dokumentieren.

3.7 Praktischer Einsatz

In den bisherigen Ausführungen wurde MDA und deren Schlüsselkonzepte in teilweise abstrakter Weise beschrieben. Wie können diese Konzepte tatsächlich in der Praxis eingesetzt werden?
An dieser Stelle wird zunächst eine grobe und vereinfachte Beschreibung des Workflows von der Modellierung bis zur Realisierung unter Einsatz von MDA vorgestellt. In Kapitel 7 folgt dann die detaillierte Dokumentation des praktischen Teils der Diplomarbeit.

In verschiedenen Artikeln [Neu03], [Völ03], [Hub03], [Boh03] bekannter Java- und OO-Zeitschriften und Dokumentationen [b+m01] ist zu lesen, dass der im einfachsten Falle dreistufige Transformationsprozess vom PIM über das PSM und dann zum Quellcode nicht praktikabel genug ist. [b+m01] schlägt deshalb einen an die Praxis angepassten Entwicklungsprozess *Generative Development Process* (GDP) auf Basis von MDA vor, in dem aus dem PIM über UML-Metamodelle direkt Quellcode erzeugt wird. Man lässt dabei die Transformation in ein PSM aus folgenden Gründen aussen vor:

- es treten Konsistenzprobleme auf, da sich Änderungen beim Round-Tripping nicht automatisch wieder in abstraktere Modelle zurückführen lassen
- manuelle Änderungen am PSM sind aufwändig, weil es näher am Quell-Code ist
-

Nach [b+m01] macht ein PSM nur dann Sinn, wenn ein Mensch direkt in das Modell eingreifen muss, um es inhaltlich zu verändern.

-

Der Einsatz von bereits vorhanden Werkzeugen und Frameworks spart erhebliche Kosten, aber man ist dafür in an deren vorgegebene Konzepte der Generierung gebunden.

ArcStyler, AndroMDA, XCoder und das b+m Generator Framework unterstützen das oben beschriebene Vorgehen, wesentliche Unterschiede sind jedoch in den verwendeten Transformationstechniken zu finden.

Allen gemein ist zunächst die Modellierung des fachlichen Designs in UML, das dem PIM entspricht. Dies geschieht entweder direkt innerhalb des Werkzeugs oder mit einem UML-Werkzeug. Vorgabe für die Modellierung ist eine abstrakte Design-Sprache, die sich aus dem Architektur-Konzept ergibt. Das Modell wird dann um die für die Transformation benötigte Semantik mit Markierungen und Stereotypen angereichert. Die Stereotypen drücken die Design-Sprache aus. Parallel dazu wird das Metamodell des entsprechenden UML-Modells erweitert, um die abstrakte Design-Sprache abzubilden und zusätzliche Funktionalität wie etwa die Validierung des Designs zu implementieren.
Für die Transformation in den Quellcode werden Templates verwendet, die bereits wiederkehrende Muster des zu erzeugenden Quellcodes enthalten. Über die Template-Sprache ist ein Zugriff auf das Metamodell möglich, so dass man abhängig davon die Transformation beeinflussen kann. Durch die Transformation wird eine Architekturrahmenimplementierung generiert, die anschließend durch manuelles Hinzufügen von Fachlogik ergänzt wird. Abschließend erfolgt die Kompilierung, das Deployment und der Test des Generats.

Stellvertretend für den Workflow integrierter Entwicklungsumgebungen und MDA Frameworks werden zwei davon kurz vorgestellt. Eine detaillierte Darstellung des in der Diplomarbeit verwendeten b+m Generator Frameworks ist in Kapitel 7 zu finden.

ArcStyler von IO Software

[ArcStyler] ist ein sehr mächtiges, kommerzielles MDA Werkzeug, das viele Aspekte von MDA berücksichtigt und einen großen Bereich des Entwicklungszyklus abdeckt. Als zentrale Komponenten verfügt ArcStyler über einen integrierten UML-Editor zum Zeichnen der UML Modelle und einen Erweiterungsmechanismus, über den sich sogenannte „Cardridges“ hinzufügen lassen. Ein Cardridge enthält die plattformspezifischen Transformationsvorschriften und Templates, über die ein Modell in ein anderes oder direkt in Quellcode überführt werden kann. Interessant ist, dass sich Cardridges selbst mit ArcStyler modellieren und anschließend generieren lassen, insofern könnte man ArcStyler auch als GeneratorGenerator bezeichnen. Cardridges sind Open Source und für die gängigen Zielplattformen wie J2EE, .NET, BEA Weblogic etc. existieren bereits Implementierungen, die mit dem Produkt mitgeliefert werden. Wenn man sich die Cardridges genauer unter die Lupe nimmt, stellt man fest, dass Modelle mit metamodell-basiertem Mapping über Templates in das Zielmodell transformiert werden. Als Template-Sprache dient die Scriptsprache JPython.

Workflow:

- Auswahl und Hinzufügen der plattformspezifischen Cartridges für die Transformation in die Zielplattformen wie z. B. J2EE, .NET, BEA Weblogic
- Erstellen des PIM als UML-Diagramm mit dem integrierten UML-Editor
- Verfeinerung des PIM durch Markierung der Modellierungselemente mit plattformspezifischen Details
- Generierung des Architekturrahmens
- Implementierung der fachlichen Funktionalität in den geschützten Bereichen des generierten Quellcodes
- Build, Deployment und Test des Quellcodes über Ant-Skript

Falls mehr Kontrolle über die Generierung notwendig sein sollte, kann man die Cartridges nach eigenen Bedürfnissen anpassen [Arc01].

b+m Generator Framework von b+m Informatik AG

Wie der Name erkennen lässt, handelt es sich um ein Framework und nicht wie z. B. bei ArcStyler um eine komplett integrierte Entwicklungsumgebung. Das Framework stellt eine Implementierung des in MOF spezifierten Metamodells für Klassen-, Zustands- und Aktivitätsdiagramme und natürlich die Generator-Engine selbst zur Verfügung. Eine sehr einfache Swing GUI zeigt die Instanziierung des Metamodells in einer Baumansicht an und ermöglicht das Starten der Transformation. Das Modellieren des PIM muss über einen externen UML-Editor erfolgen. Als Eingabe für den Generator dient das serialisierte PIM im XMI Format. Der Transformationsprozess läuft metamodell-basiert über einen Template-Mechanismus ab. Als Template-Sprache dient XPand, eine einfache, proprietäre und an den Generator angepasste Skript-Sprache.

Workflow

- Erstellen einer Referenzimplementierung
- Design-Muster identifizieren und Designsprache im Metamodell abbilden
- Generator-Templates ableiten und in der Template-Sprache implementieren
- Erstellen des PIM als UML-Diagramm mit dem integrierten UML-Editor
- Auszeichnung des PIM durch Markierung der Modellierungselemente mit Tagged Values und Stereotypen, die durch die Designsprache festgelegt sind
- Generierung des Architekturrahmens
- Implementierung der fachlichen Funktionalität in den geschützten Bereichen des generierten Quellcodes
- Build, Deployment und Test des Quellcodes über Ant-Skript

Eine sehr ähnliche Vorgehensweise ist bei [AndroMDA] zu finden, einem Open Source Framework, das aus UML2EJB enstanden ist. Statt einer proprietären Skriptsprache für die Templates wird jedoch Velocity von Apache.org und XDoclet verwendet.

Fazit

Wenn man sich die Werkzeuge anschaut, stellt man fest, dass die template-gestützte Generierung mithilfe einer Skriptsprache und Metamodellen als Navigationsmöglichkeit bevorzugt wird, obwohl MDA auch andere Alternativen vorschlägt. Es fällt auch auf, dass recht unterschiedliche Skript-Sprachen verwendet werden, was zu einer starken Bindung der Entwickler an ein bestimmtes Werkzeug führen kann.

In der folgenden Liste sind nochmals die Gemeinsamkeiten der Werkzeuge zusammengefasst.

- grafische Modellierung der Architektur als UML Modell mit einem Modellierungswerkzeug
- Verfeinerung des Modells durch Auszeichnung der einzelnen Modellierungselemente mit Markierungen und Stereotypen nach einem UML Profil oder Design-Sprache
- Erweiterung des Metamodells zur Abbildung der Designsprache oder des UML Profils
- Implementierung des Mappings durch Templates und programmatische Konstrukte innerhalb der Metamodellerweiterung für die jeweilige Zielplattform. Dies kann durch Out-of-the-Box Komponenten (Plugins, Cartridges) bereits integrierter Bestandteil des Werkzeugs sein

- Serialisierung des Modells in XMI
- metamodell- und templategestützte Transformation des Modells in einen Implementierungsrahmen
- Implementierung der Fachlogik in geschützten Bereichen des Quellcodes
- Build, Deployment und Test der Realisierung mit Hilfe eines Ant Build Skripts
-

4 Java 2 Platform Enterprise Edition (J2EE)

J2EE ist ein von Sun Microsystems definierter Standard für die Entwicklung von komponentenbasierten, mehrschichtigen Unternehmensanwendungen. Zu J2EE gehören unter anderem Java Server Pages (JSP) für die Entwicklung von web-basierten Frontends und die JavaBeans bzw. Enterprise Java Beans Technologie zur Umsetzung der Schichten des Backends. Als weitere Features sind Web Services und Entwicklungswerkzeuge (SDK) zu nennen.
J2EE wird als Zielplattform für die Generierung des Implementierungsrahmens in der Diplomarbeit verwendet.

4.1 Überblick

J2EE ist neben Microsoft .NET als konkurrierender Plattform für unternehmensweit verteilte Anwendungen zur Implementierung von Businessfunktionalität eine breit unterstützte Technologie, zumal sie viele APIs und ein robustes Konzept bietet. Das Konzept basiert auf sogenannten „Containern", abstrakten Komponenten, die Teilfunktionalitäten gruppieren und voneinander trennen und die Laufzeitumgebung für J2EE-Komponenten (Servlets, Enterprise JavaBeans usw.) zur Verfügung stellt. J2EE definiert insgesamt vier Container [Sha03]:

- *Application Client Container*: Java Applikationen, die auf dem Client-Rechner ausgeführt werden (Fat-Clients)
- *Applet Container*: Applets sind spezielle Java Applikationen, die typischerweise als Objekt in eine HTML-Seite eingebettet sind und vom Web-Browser gestartet werden
- *Web Container*: Java Server Pages und Servlets sind Web-Komponenten, die durch einen HTTP-Request des Clients von einem Web Application Server ausgeführt werden und als Ergebnis HTML-Seiten an den Client zurückliefern
- *EJB Container*: Enterprise JavaBeans werden in einer Umgebung auf dem Application Server ausgeführt, die Transaktionen unterstützt.

Alle vier Container verwenden das Java Development Kit als Implementierungsgrundlage.

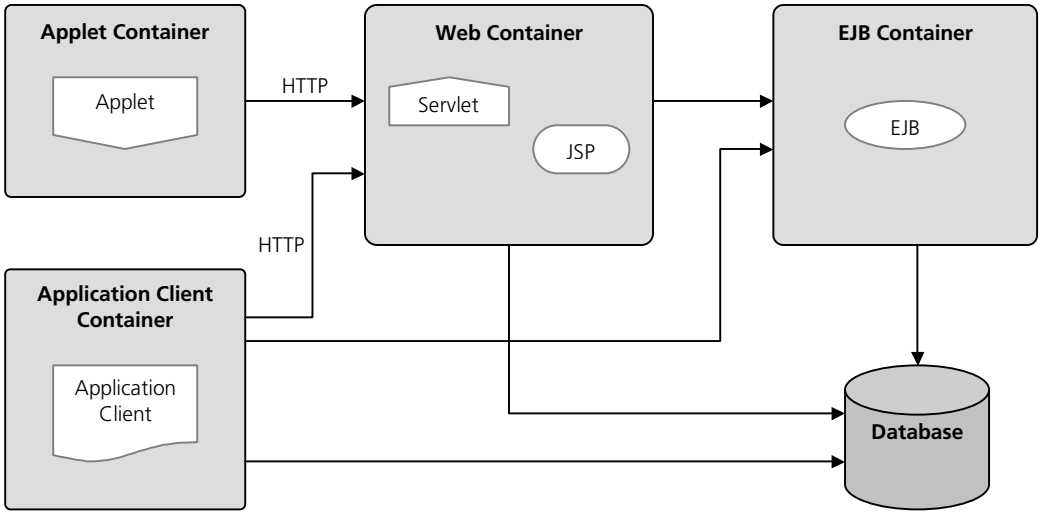

Abbildung 8: Die J2EE Architektur

J2EE erfordert in den meisten Fällen ein Datenbanksystem für die Persistierung von Daten über die Java Database Connectivity (JDBC) API.
Eine J2EE Anwendung ist skalierbar und komponentenorientiert. Eine J2EE Anwendung setzt sich aus einem oder mehreren J2EE Modulen zusammen. Ein J2EE Modul besteht wiederum aus einer oder mehreren J2EE Komponenten, wie etwa EJB oder einer Web Application Komponenten. Rein physikalisch wird diese Struktur in einer hierarchischen Archiv-Datei abgebildet, die an den Application Server ausgeliefert werden kann.

Das Konzept des EJB Containers und der EJB Komponenten wird im folgenden näher beschrieben, da die in der Diplomarbeit aus dem PIM generierte Architektur auf EJB Komponenten basiert. Die Beschreibung von EJB beschränkt sich auf die für das Verständnis der Architektur wichtigen Komponenten. Detaillierte Informationen sind in [Rom02] und [Dem03] zu finden.

4.2 Enterprise JavaBeans (EJB)

Die Enterprise JavaBeans Architektur ist eine komponentenbasierte Architektur zur Entwicklung von unternehmensweiten verteilten Business-Systemen. Sie ist Bestandteil von J2EE. Anwendungen werden innerhalb des EJB Containers ausgeführt, sind skalierbar, transaktionsgestützt und Multi-User fähig. Nach [OMG01] ist eine Komponente eine modulare Einheit, die wohldefinierte Schnittstellen besitzt und innerhalb ihrer Umgebung austauschbar ist. Die EJB Spezifikation bezeichnet Enterprise JavaBeans als Komponenten (hier im folgenden *Beans* genannt), auf die die genannte Beschreibung zutrifft. Eine Bean ist als Black-Box

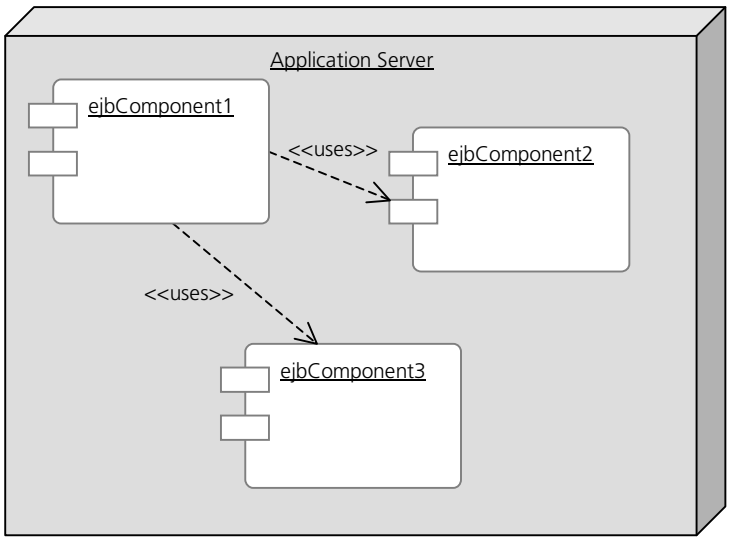

Abbildung 9: EJB Komponenten im EJB Container eines Application Servers.

zu sehen, die fachliche Funktionalität kapselt. Beans sind Bestandteil einer EJB Anwendung.
Ein zentrales Ziel des EJB Standards ist es, komplexe Systeme aus verschiedenenen Beans zusammenbauen zu können. Als Vision soll ein Markt entstehen, in dem Komponenten angeboten, gekauft und wiederverwendet werden können [Rom02]. Für den Anwendungsentwickler stellt Sun das Package `javax.ejb` zur Verfügung, das Bestandteil der J2EE API ist. Das Package enthält einen Satz von Interfaces und Exceptions, die der Bean Entwickler implementieren muss, um eine EJB konforme Architektur zu erzeugen.

4.2.1 EJB Container

Der EJB Container ist die Laufzeitumgebung für Beans. Die EJB Spezifikation sieht vor, dass der Container vom sogenannten „Container und Server Provider" zur Verfügung gestellt wird. In der Regel ist er Bestandteil des Application Servers, der als kommerzielles oder freies Produkt vertrieben wird. JBoss, BEA Weblogic oder IBM Websphere Application Server sind Produkte, die einen EJB Container implementieren.

Der Container bildet den Kontext und die Laufzeitumgebung für die darin enthaltenen Beans. Eine Bean kann über den Container mit anderen Beans kommunizieren. Analoges gilt für Klienten, die von aussen auf Beans zugreifen möchten. Technische Funktionalität wie Middleware Services (Transaktionsmanagement, Security, Threading etc.) wird vor den Beans gekapselt und ist im Container selbst implementiert.

4.2.2 EJB API

Die EJB Spezifikation von Sun stellt einen Satz von Java Interfaces und Exceptions zur Verfügung (vgl. Abbildung 10), die von Bean Entwicklern erweitert oder implementiert werden müssen, um eine J2EE konforme und im EJB Container lauffähige Version zu erhalten.

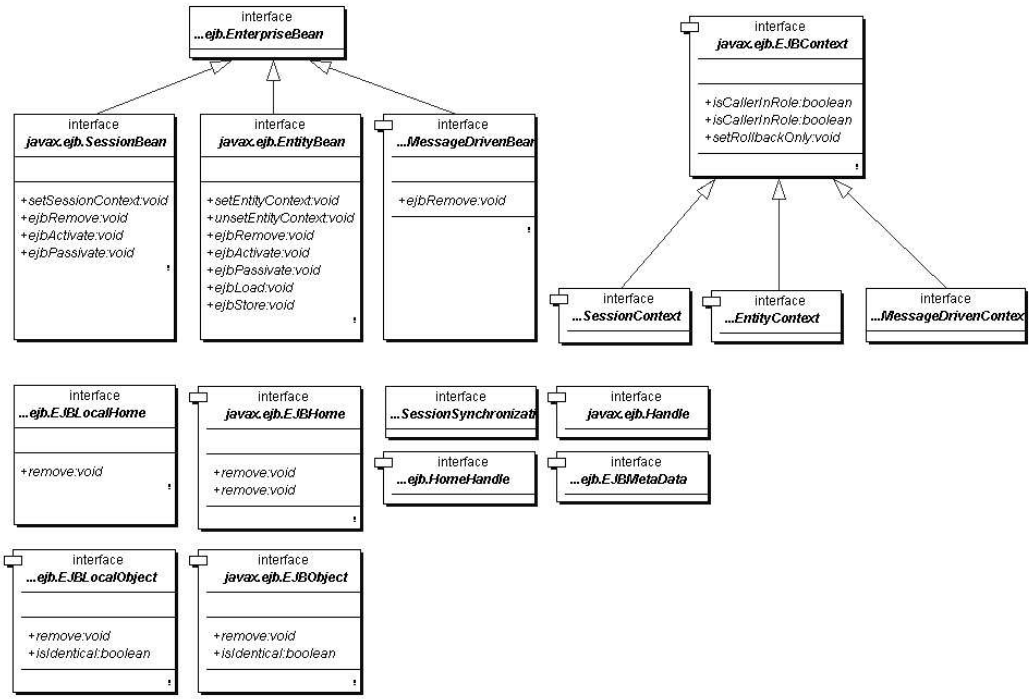

Abbildung 10: Das Klassendiagramm der EJB API 1.4 ohne Exception-Klassen

Enterprise Bean

Eine Enterprise Bean ist eine server-seitige Softwarekomponente, die in eine mehrschichtige verteilte Umgebung installiert werden kann. Sie kann aus einer oder mehreren Java Klassen und Interfaces bestehen. Klienten oder andere Beans kommunizieren immer über das Interface der Komponente, das gemäß der EJB Spezifikation bestimmte Methoden zur Verfügung stellen muss. Da Enterprise Beans innerhalb einer Anwendung verschiedene Aufgaben erledigen können, teilt EJB 2.0 sie weiter auf:

- ***Session Beans*** modellieren fachliche Prozesse oder Aktivitäten. Sie sind vergleichbar mit Verben in der deutschen Grammatik, sie führen eine Aktion aus. Das Einzahlen oder Abheben eines bestimmten Geldbetrags auf ein Bankkonto würde man als Session Bean implementieren.
- ***Entity Beans*** modellieren fachliche Daten. Sie entsprechen den Substantiven in der deutschen Grammatik. Um beim erwähnten Bank-Beispiel zu bleiben – das Konto oder die Bank werden als Entity Bean implementiert.

Innerhalb des Packages `javax.ejb` werden einige Interfaces definiert, die mit einer Klasse implementiert und durch Subinterfaces erweitert werden können.

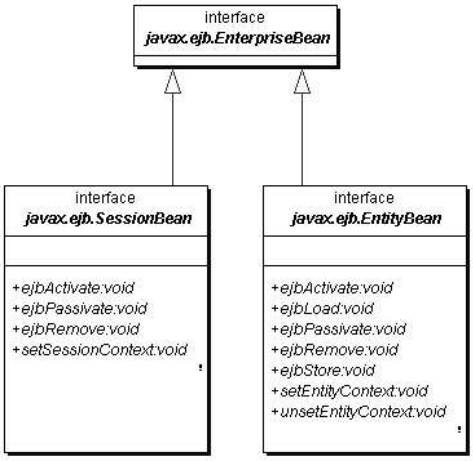

Abbildung 11: SessionBean und EntityBean Interfaces.

EJB Object

Beans interagieren nie direkt mit anderen Beans, sondern über eine dazwischen liegende Schicht. Der Container verwendet Objekte vom Typ EJBObject, um Aufrufe an Beans zunächst abzufangen, zu bearbeiten und anschließend an die Bean weiterzuleiten (Interception-Mechanismus). Der Grund für diese Art der Implementierung ist die Kapselung der Middleware-Services vor der durch den Bean Entwickler implementierten Fachlogik. Jede fachliche Methode einer Bean muss dabei jedoch durch das EJBObject gespiegelt werden, ansonsten wäre einem der Zugriff auf die Funktionalität verwehrt. Salopp kann man ausdrücken, dass jeder Bean Klasse ein „Bodyguard“ oder „Agent“ zugeordnet ist, den man vor einer Anfrage konsultieren muss, bevor man durchgelassen wird.

Home Object

Wenn Klienten also nie direkt auf Beans zugreifen dürfen, stellt sich die Frage, woher man dann Instanzen vom Typ EJBObject bekommt. EJB verbietet die direkte Instanziierung (mit new) von EJBObject, da EJB Objects auf verschiedenen Maschinen existieren können. EJB be-

dient sich deshalb des Fabrikmusters und der Fabrikmethoden [Gam94], um neue EJB Objects zu erzeugen. Die Spezifikation sieht dafür eine Implementierung des `javax.ejb.HomeObject` Interfaces vor. Dessen Aufgabe ist nicht nur das Erzeugen, sondern auch das Finden und Löschen von EJB Objects.
Home Objects enthalten übrigens containerspezifische Logik, können also abhängig vom Server Hersteller unterschiedlich implementiert sein.

Home Interface

Um nun der Fabrik für EJB Objects mitzuteilen, wie sie denn genau neue EJB Objects erzeugen soll, kann der Bean Entwickler das Interface `javax.ejb.EJBHome` um eigene Fabrikmethoden erweitern. Wenn ein EJB Object beispielsweise mit zwei Parametern erzeugt werden soll, erweitert man das Home Interface und fügt eine neue Fabrikmethode hinzu, die die beiden Parameter in der Signatur enthält. Der Container sorgt dann dafür, dass dieses Interface durch das Home Object implementiert wird. Ein Klient sieht hier also nur das Home Interface und nie das Home Object selbst.

Remote Interface

Für jede Bean muss der Bean Entwickler das Interface `javax.ejb.EJBObject` erweitern und mit den fachlichen Methoden der Bean anreichern. Das vom Container erzeugte EJB Object implementiert dieses Interface und ermöglicht somit die Delegation der fachlichen Methodenaufrufe an die Bean.
Ein Klient ruft also zunächst die fachliche Methode des Remote Interface auf. Da sich hinter dem Interface das vom Container generierte EJB Object verbirgt, wird der Aufruf an die entsprechende Bean weitergeleitet.

Local Interface

Objekte in verteilten System können über das ganze Netzwerk verteilt sein und in verschiedenen Server Prozessen laufen. Um das Auffinden von Objekten und Methodenaufrufe (`create()`, `remove()`) auf den Objekten zu ermöglichen, wird der in [Rom02] beschriebene Finder-Mechanismus verwendet. Jeder Aufruf auf einem verteilten Objekt kostet durch (De-) Serialisierung und Übertragung über das Netzwerk Zeit, was die Performance für den Anwender unter Umständen beeinträchtigt. Befinden sich jedoch die Beans auf einer Maschine in einem Prozess, wäre die Serialisierung und Übertragung nicht notwendig.
Eine Abhilfe schaffen die seit EJB 2.0 möglichen Local Interfaces, die performante und effiziente Aufrufe von Beans ermöglichen. Der Unterschied liegt für den Bean Entwickler darin, dass er statt `javax.ejb.EJBObject` das Interface `javax.ejb.EJBLocalObject` erweitern muss (analoges gilt für das Home Interface).

Session Bean

Session Beans sind Objekte, die fachliche Prozesse ausdrücken. Sie können fachliche Logik und Algorithmen enthalten und einen Workflow abbilden. Der Hauptunterschied zwischen einer Session Bean und einer Entity Bean liegt in der Lebenszeit. Eine Session Bean „lebt“ solange, wie die Session des Klienten besteht. Wenn also der Klient beispielsweise das Browser Fenster einer EJB Anwendung schließt, wird die Session Bean gelöscht.
Im Gegensatz dazu können Entity Beans über Wochen und Monate hinweg „leben“, da sie persistente Objekte darstellen, d. h. sie werden auf einem permanten Speichermedium gespeichert.

EJB unterscheidet zwischen sogenannten Stateful und Stateless Session Beans. Eine Stateful Session Bean kann über die Dauer einer Session einen an den Klienten gebundenen Zustand speichern, der für die Ausführung des fachlichen Prozesses nützlich ist.

Für die Implementierung einer Session Bean sind folgende Schritte notwendig

- Erweiterung des Remote Interfaces `javax.ejb.EJBObject` mit den Signaturen der fachlichen Methoden
- Hinzufügen von fachlichem Code in einer Klasse, die das erweiterte Remote Interface und das Interface `javax.ejb.SessionBean` implementiert
- Erweiterung des Home Interfaces `javax.ejb.EJBHome` mit den Signaturen der Fabrikmethoden für den Lifecycle der Bean

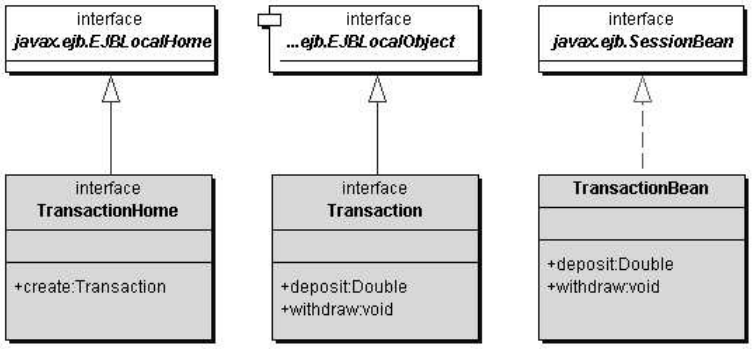

Abbildung 12: Session Bean Komponente

Entity Bean

Entity Beans modellieren die fachlichen Daten, die in einer Anwendung persistiert werden müssen, in objektorientierter Weise. Eine Entity Bean speichert Werte in Form von Attributen, auf die über getter und setter Methoden zugegriffen werden kann. Für die Persistierung wird ein permanentes Speichermedium verwendet, in der Regel eine relationale Datenbank.
Da eine relationale Datenbank Daten als Einträge in Tabellen ablegt, stellt sich das Problem des objekt-relationalen Mappings (O/R). Wie werden Objekte in einer Datenbank abgebildet? Solange eine objektorientierte Datenstruktur keine Vererbung verwendet, ist das Mapping straightforward: Klassen werden als Tabellen und Attribute als Spalten abgebildet. Objekte speichern Attributwerte zur Laufzeit und entsprechen einem Tabelleneintrag. Assoziative Beziehungen zwischen Objekten werden auf der Datenbank durch Primär- und Fremdschlüssel Beziehungen ausgedrückt. Die EJB Architektur verwendet diesen Ansatz und hält die Struktur von Entity Beans bewusst flach, um die Persistierung von Vererbung zu umgehen.
Eine Entity Bean repräsentiert eine Tabelle und Instanzen einer Entity Bean werden als Einträge in dieser Tabelle abgespeichert. Eine Entity Bean ist jedoch nicht nur eine Komponente, die in ihren Attributen Daten speichert, sondern sie kümmert sich auch um die Persistierung dieser Daten.
Entity Beans müssen eindeutig sein, da sie ansonsten in einem verteilten System nicht identifizierbar und auffindbar wären. Aus diesem Grund ist jeder Bean eine Primary Key Klasse zugeordnet, die einen eindeutigen Schlüsselwert erzeugt.

Für die Implementierung einer Entity Bean sind wie schon bei der Session Bean folgende Schritte notwendig

- Erweiterung des Remote Interfaces `javax.ejb.EJBObject` mit den Signaturen der getter und setter Methoden der fachlichen Attribute
- Hinzufügen von fachlichen Attributen in Form von abstrakten getter und setter Methoden in einer abstrakten Klasse, die das Entity Bean Interface `javax.ejb.EntityBean` implementiert
- Erweiterung des Home Interfaces `javax.ejb.EJBHome` mit den Signaturen der Fabrikmethoden für den Lifecycle der Bean
- Implementierung der Primay Key Klasse

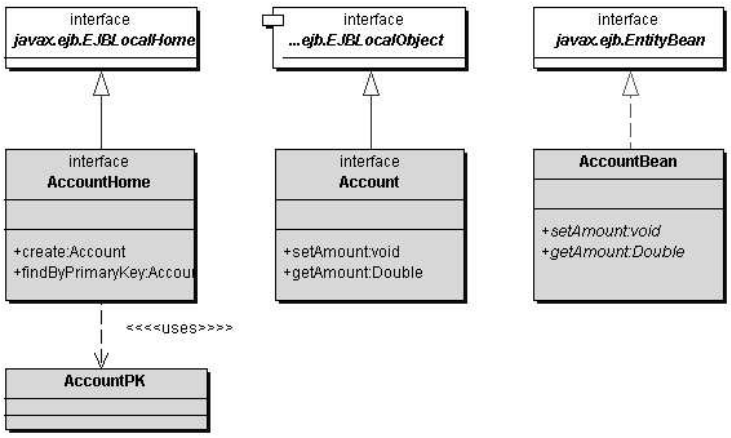

Abbildung 13: Entity Bean Komponente

4.2.3 Zusammenbau und Auslieferung

Sobald Enterprise Beans durch den Entwickler implementiert sind, müssen sie in irgendeiner Weise und Form zu einer Enterprise Anwendung zusammengebaut (Assembly) und an den Application Server ausgeliefert werden (Deployment). Rein physikalisch bedeutet dies, dass sich die Beans in einem Verzeichnis befinden müssen, das dem Server bzw. dem EJB Container zugänglich ist, um sie beim Start laden zu können.

EJB Deployment Deskriptoren

Um dem Container Informationen über die ausgelieferten EJB Komponenten, deren Abhängigkeiten und die benötigten Einstellungen für die Middleware Dienste mitzuteilen, definiert EJB einen abstrakten Deployment Deskriptor in Form einer XML Datei ejb-jar.xml. Einträge im Deployment Deskriptor erfolgen in einem durch die EJB Spezifikation vorgegebenen XML Dialekt, der unter anderem die folgenden Top Level Elemente enthält:

- `<enterprise-beans>`: Beschreibt alle Beans der Komponente. Bei Entity Beans werden die persistenten Attribute mit angegeben
- `<relationships>`: Beschreibt die Assoziationen zwischen Entity Beans (Multiplizität, Rollen)
- `<assembly-descriptor>`: Beschreibt die Security Rollen, die den Zugriffsschutz auf Methoden festlegen

Der Deployment Descriptor beschreibt die Komponenten in herstellerunabhängiger und abstrakter Weise.
Die EJB Spezifkation sieht vor, dass EJB Container Hersteller proprietäre Features (wie Datenbankmapping, Load-Balancing, Clustering, Monitoring etc) hinzufügen können. Diese werden in einer oder mehreren separaten Deployment Descriptor Dateien definiert, die den bereits beschriebenen Deskriptor ergänzen. BEA Weblogic verwendet beispielsweise zwei technische Deskriptoren, einer davon beschreibt das Mapping der Beans auf die Struktur der Datenbank, der andere allgemeine Weblogic-spezifische Konfigurationseinstellungen.

Application Deployment Deskriptor

Eine J2EE Anwendung kann aus mehreren Komponenten bestehen. Die Beschreibung dieser Komponenten wird dem Container über den Application Deployment Descriptor mitgeteilt. Der Deskriptor ist eine XML Datei, in der die einzelnen Komponenten als „Module" angegeben werden und die sich direkt in einem Enterprise Application Archiv (vgl. Abbildung 14) befindet.

Zusammenbau/Paketierung

In einem Software-Projekt, das EJB Technologie nutzt, entstehen viele Artefakte wie Klassen, Interfaces und Deskriptor Dateien. Für die Auslieferung einer Anwendung wäre es recht unpraktisch, alle diese Artefakte einzeln transportieren zu müssen. Manche lassen sich auch logisch zu einer abgeschlossenen Einheit (z. B. als wiederwendbare Bibliotheken oder Subsysteme) zusammenfassen und sollten dementsprechend auch verwendbar sein.
Aus diesen Gründen bietet J2EE eine zweistufige Paketierungsmöglichkeit an. Einzelne logisch zusammengehörende Enterprise Bean Komponenten können mit dem Deployment Deskriptor zusammen in ein EJB-JAR Archiv Datei gepackt werden. Das Archiv entspricht einem J2EE Modul. Mehrere dieser Module können in der zweiten Stufe nochmals in ein EAR Archiv mit einem Application Deployment Deskriptor gespeichert werden, so dass die gesamte J2EE Anwendung aus insgesamt einer Archiv-Datei besteht, die ausgeliefert werden kann. Für den Klienten bestimmte Dateien können entsprechend in einem Web Application Archiv (WAR) zusammengefasst werden.

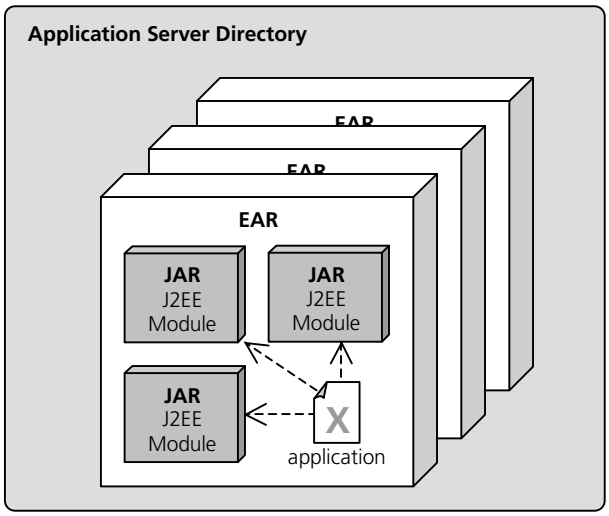

Abbildung 14: Das EAR Archiv.

4.2.4 Persistenz-Mechanismen

Da Entity Beans als Objekte im Speicher vorliegen, müssen sie über einen Mechanismus persistent in der Datenbank abgelegt werden, da ansonsten die Daten verloren gehen würden. Für die Persistierung bietet EJB zwei Möglichkeiten an: Bean Managed Persistance (BMP) und Container Managed Persistance (CMP).

Bean Managed Persistance (BMP)

Bei einer Bean-Managed Persistance Bean muss der Bean Entwickler sich selbst um das Abspeichern in der Datenbank kümmern, d. h. er nutzt die JDBC API, um Persistenzoperationen wie Laden, Speichern und das Suchen von Entity Beans zu schreiben. Der Code wird in der Regel in die Entity Bean selbst eingefügt.

Container Managed Persistance (CMP)

Mit EJB ist es möglich, Persistenzoperationen vom Container selbst ausführen zu lassen. Der Container generiert den notwendigen Datenbankzugriffscode aus den vorhanden Entity Beans. Die Generierung kann in deklarativer Weise im Deployment Deskriptor beeinflusst werden.

Aus CMP ergeben sich daher folgende Vorteile:

- Entity Beans werden nicht durch rein technischen Code "verunreinigt" und bleiben dadurch klein und gut lesbar
- die Implementierung übernimmt der Server Hersteller und erspart dem Entwickler dadurch viel Zeit
- man ist nicht an die Art der Datenbank (relational, objektorientiert) gebunden
- nach [Rom02] ist CMP in den meisten Fällen performanter als BMP, da z. B. Persistenz Operationen in optimierten SQL Abfragen zusammengefasst werden

CMP hat jedoch auch Nachteile:

- man verliert die vollständige Kontrolle über den Datenbank-Zugriff
- Fehler in der CMP Implementierung des Containers können schwer zu finden sein
- man ist abhängig vom Server Hersteller
-

In der Diplomarbeit wird aufgrund der oben genannte Vorteile ausschließlich CMP als Persistenzmechanismus verwendet.

Container Managed Relations (CMR)

Wenn man nach EJB 1.1 auf Beziehungen zwischen einzelnen EntityBeans zugreifen wollte, musste man entsprechende Finder-Methoden im Remote Home Interface der EntityBean definieren und im Deployment Deskriptor EJB-QL Anweisungen notieren (eine SQL-ähnliche Skriptsprache), die über den jeweiligen Primärschlüssel verbundene EntityBeans gesucht haben. Mit CMP ist es nun auch möglich, Beziehungen zwischen EntityBeans im abstrakten Deployment Deskriptor aus logischer Sicht zu deklarieren und den Zugriff durch den EJB Container ausführen zu lassen.
In den EntityBeans selbst werden dabei durch den Bean Entwickler abstrakte getter und setter Methoden definiert, die die jeweilig verbundenen EntityBeans als Referenz zurückgeben oder speichern. Bei einer 1:1 Beziehung wird ein Objekt des EntityBean Typs zurückgegeben, bei einer 1:N Beziehung eine `java.util.Collection` (bzw. `java.util.Set`) , die mehrere EntityBean Objekte enthält.

Im abstrakten Deployment Deskriptor erfolgt die Deklaration von Beziehungen im `<relationships>` Element. Dort wird für jede Beziehung ein `<ejb-relation>` Element eingefügt, in dem weitere Eigenschaften wie Rollennamen, Multiplizität und der Name des Attributs angegeben werden, das die Referenz auf eine EntityBean oder eine Collection von EntityBeans speichert. Direktionalität kann durch das Weglassen des Attributs in der jeweiligen Rolle ausgedrückt werden.
In den technischen Deskriptoren des Container Herstellers (wie beispielsweise BEA Weblogic) muss aus den Informationen des abstrakten Deskriptors ein Mapping auf die physikalischen Gegebenheiten der Datenbank erfolgen.

4.2.5 Beziehungen

Beziehungen beschreiben den Zusammenhang zwischen zwei oder mehreren Entitäten (Sonderfall: Beziehung einer Entität zu sich selbst). Beziehungen zeichnen sich unter anderem durch Eigenschaften wie Kardinalität und Direktionalität aus. Die Kardinalität gibt an, wieviele Instanzen auf beiden Seiten der Beziehung zur Laufzeit vorhanden sein dürfen.
Dabei unterscheidet man zwischen

- **1:1** Beziehungen, Bsp.: Ein Mitarbeiter hat genau eine Adresse
- **1:N** Beziehungen, Bsp.: Ein Teamleiter hat mehrere Mitarbeiter
- **N:M** Beziehungen, Bsp.: ein Mitarbeiter kann in mehreren email-Verteiler Listen eingetragen sein und umgekehrt können jeder Liste mehrere Mitarbeiter zugeordnet sein

Die Direktionalität bestimmt die Navigierbarkeit bzw. die Sichtbarkeit der Entitäten untereinander. Man unterscheidet zwischen

- unidirektionalen
- bidirektionalen

Beziehungen. Bei einer unidirektionalen Beziehung kann man von einer Entität A zu einer Entität B navigieren (A „kennt“ B), jedoch nicht umgekehrt. Bei einer bidirektionalen Beziehung sind beide Richtungen möglich (A „kennt“ B und B „kennt“ A).
Auf Datenbankebene werden Beziehungen zwischen Entitäten über Fremdschlüsselfelder ausgedrückt, die Primärschlüsselwerte von anderen Entitäten speichern, wohingegen auf Objektebene Attribute verwendet werden, die Referenzen auf andere Objekte speichern.
Für die Implementierung von Beziehungen bietet EJB wiederum zwei Möglichkeiten an: Bean-Managed Relationships (BMR) oder Container-Managed Relationships (CMR). Bei ersterem ist der Entwickler selbst gefordert, den Zugriff auf Beziehungen mit der JDBC API oder EJB-QL (Query Language) in den Beans zu programmieren. CMR wird wie schon CMP deklarativ in den Deployment Deskriptoren konfiguriert und dann durch den Container generiert. Die generierte Architektur in der Diplomarbeit verwendet CMR und zeigt Beispiele für Beziehungen zwischen Entitäten mit den Kardinalitäten 1:1 und 1:N sowie Uni- und Bidirektionalität.

4.2.6 Generierung

EJB Architekturen zeichnen sich durch wiederkehrende Implementierungs- und Strukturmuster und Redundanzen in Namensdeklarationen aus. So sind für jedes Enterprise Bean zwei Interfaces notwendig, in denen sich beispielsweise die Methodensignaturen wiederholen. In den Deployment Deskriptoren wiederholen sich die Namen der Beans. Wer einmal mehrere Beans von Hand entwickelt hat, stellt fest, dass es eine sehr mühsame Arbeit ist, ausser der Fachlogik noch die durch die EJB Architektur vorgegebenen Schnittstellen anzulegen. Ändert

sich der Name einer Klasse oder werden neue Fachmethoden hinzugefügt, so muss dies an mehreren Stellen nachgezogen werden, von der höheren Fehlerquote ganz zu schweigen. In Projekten mit vielen Anwendungsfällen und Datenbankentitäten, die als Entity Beans abgebildet werden müssen, wird dieses Vorgehen zu einer großen Last. Aus diesem Grund sind EJB Komponenten ein eindeutiger Fall für den Einsatz eines Quellcode-Generators, der einem diese mühsame Arbeit abnimmt und immer wiederkehrende Codesegmente generiert.

Obwohl in der MDA wie bei herkömmlichen EJB Code-Generatoren schlussendlich auch Code generiert wird, abstrahiert MDA von der technischen Implementierung, bedient sich jedoch in der Regel der Code-Generatoren, um die Transformation in ein plattformabhängiges Modell auszuführen. In der MDA wird der Generator metamodell-basiert angesteuert, während ein EJB Code-Generator beliebigen proprietären Input erwartet.

5 BMW Component Architecture (CA)

Die in der Diplomarbeit verwendeten PIMs orientieren sich an den Vorgaben und den Begriffen der BMW Component Architecture (CA). Die CA gibt die Designsprache für die PIMs vor, was sich in den Erweiterungen des Metamodells ausdrückt, das den PIMs zugrunde liegt.

5.1 Überblick

Die CA [Web02] ist eine von IBM für BMW entwickelte System-Architektur, um ein einheitliches Konzept für verteilte Anwendungen vorzugeben. Sie ist technologie- und plattformunabhängig und entspricht somit einem architekturzentriertem PIM in der MDA. Sie definiert zudem ein Mapping nach J2EE. Der Hauptnutzen der CA liegt darin, Software-Architekten eine Grundlage für die Entwicklung konkreter Architekturen an die Hand zu geben, um das Rad für jedes Projekt nicht nochmal neu erfinden zu müssen [Web02]. Komponenten, die nach der CA entworfen wurden, implementieren fachliche Funktionalität und werden auf einem Application Server ausgeführt. Web-basierte Komponenten der Präsentationsschicht sind nicht Gegenstand der CA.

5.2 Design

Das Design der CA ist so ausgelegt, dass es den typischen Anforderungen von verteilten Software-Systemen gerecht wird. Es folgt eine Auflistung einiger wichtiger Aspekte

- Trennung der externen, fachlichen Server Schnittstelle von der technischen Implementierung
- Ermöglichung verschiedener Sichten (Views) auf Funktionalität und Daten (Model)
- klare Trennung von Schichten der Fachlogik
- Unterstützung von MDA
- Wiederverwendbarkeit von Komponenten
- Skalierbarkeit: Clustering, Load-Balancing
- Sicherheit
- …

Die CA definiert und benennt größere unabhängige Einheiten einer Anwendung als Business Components. Business Components entsprechen der in [OMG01] spezifizierten Semantik der Komponente. Sie stellen nach aussen hin ein fachliches Interface zur Verfügung, können in unterschiedlichen Granularitäten auftreten und sind innerhalb einer Schicht gleich strukturiert.

Beziehungen zwischen Business Components müssen unidirektional sein, d. h. eine Komponente kann eine andere verwenden, aber nicht umgekehrt.

5.3 Kernkonzept

Das Kernkonzept basiert auf Best Practices und Mustern wie sie etwa in [Mar02] durch die EJB Design Patterns beschrieben werden. Es fällt auf, dass es zwischen der CA und der EJB Architektur große Ähnlichkeit gibt, was ein späteres Mapping sehr erleichtet. Die CA abstrahiert jedoch in der Namensgebung und gibt gewisse Einschränkungen (Constraints) des Entwurfs vor.

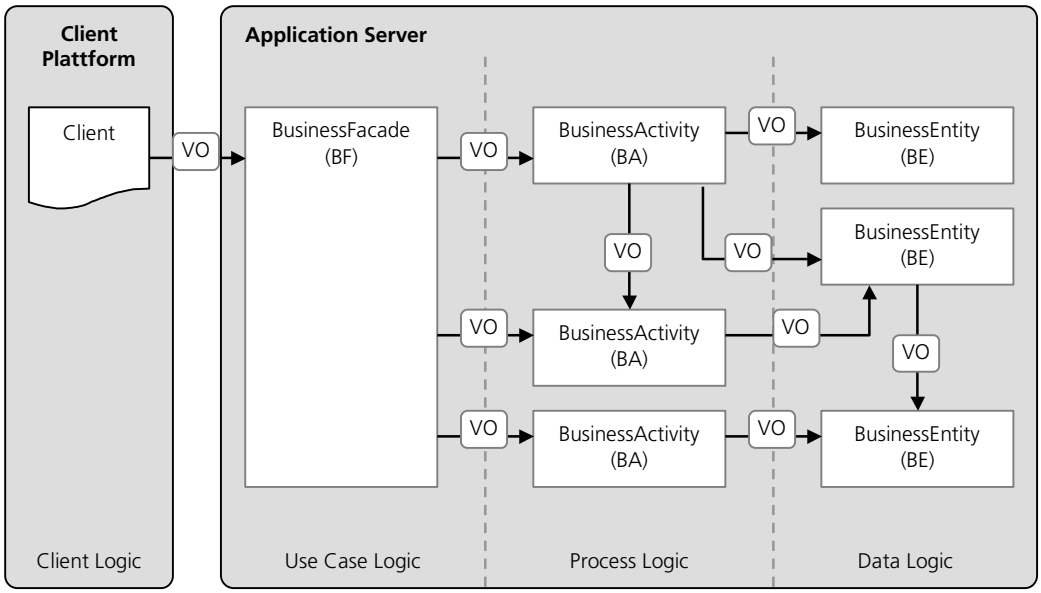

Abbildung 15: BMW Component Architecture (CA).

5.3.1 Business Object (BO)

Business Objects bilden den abstrakten Kern der CA und kapseln die Fachlogik. Sie sind mit den Enterprise Beans in der EJB Plattform vergleichbar. Da Business Objects verschiedene Aufgaben in einem System übernehmen können, unterscheidet man folgende Subtypen

- Business Facade (BF): die externe Schnittstelle des Application Servers
- Business Activity (BA): implementiert Business Prozesslogik
- Business Entity (BE): implementiert die Datenzugriffslogik

Für den Datenaustausch zwischen Business Objects werden Value Objects verwendet.

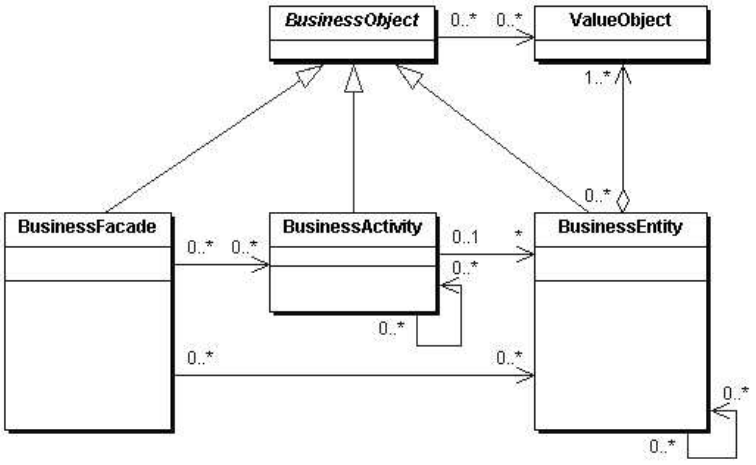

Abbildung 16: Das Klassendiagramm der CA (Auszug)

5.3.2 Business Facade (BF)

Die Business Facade ist die öffentliche Schnittstelle einer verteilten Anwendung. In anderen Architekturen wird auch der synonyme Begriff des Use Case Controllers verwendet. Die Business Facade stellt Operationen zur Verfügung, die fachliche Transaktionen auf dem Server ausführen. Für die Ausführung eines Anwendungsfalls wird in der Regel auf mehrere serverseitige Objekte unterschiedlicher Granularität zugegriffen, was zu mehreren Netzwerkzugriffen (und möglicherweise Transaktionen) und damit zu Performance Einbußen führen kann. Business Facades fassen die verschiedenen Zugriffe in einer Transaktion zusammen und verstecken die komplexe, server-seitige Logik durch eine flache und einheitliche Schnittstelle vor dem Klienten. Typischerweise bildet sie den Workflow der Anwendungsfälle ab, implementiert diese jedoch nicht, sondern delegiert an die Business Activities.
Business Facades haben Zugriff auf Business Activities und Business Entities, jedoch nicht auf andere Business Facades.

5.3.3 Business Activity (BA)

Business Activities enthalten die Fachlogik und sind dem Befehlsmuster vergleichbar [Gam94]. Sie bilden die feingranularen Aktivitäten aus dem fachlichen Analysemodell ab. So kann z. B. das Überprüfen der Benutzerrechte oder die Validierung und Plausibilität von Eingabedaten in einer Business Activity implementiert werden. Business Activities werden von der Business Facade aufgerufen. Business Activities sollten so implementiert werden, dass sie wiederverwendbar sind, d. h. sie dürfen keine Annahmen über den Kontext des sie umgebenden Systems machen.
Business Activities haben Zugriff auf andere Business Activities oder auf Business Entities, jedoch nicht auf Business Facades.

5.3.4 Business Entity (BE)

Business Entities speichern die fachlichen Daten einer verteilten Anwendung und machen sie für die Business Facade und Business Activities zugreifbar. Sie führen keine oder nur äusserst eingeschränkt Fachlogik aus. Sie sind veranwortlich für die Persistierung der Daten auf der Datenbank. Sie haben unidirektionalen Zugriff auf andere Business Entities, jedoch nicht auf Business Activities oder Business Facades.

5.3.5 Value Object (VO)

Ein Value Object ist ein leichtgewichtiger Datencontainer, der fachliche Daten für den Transport zwischen Klient und Server speichert. Value Objects werden durch Business Entities persistent gemacht. Über Value Objects kann man Daten vom Klienten zum Server austauschen, ohne mehrere Netzwerkzugriffe zu machen. Der Klient erzeugt ein Value Object und übergibt es an die fachlichen Operationen der Business Facade. Diese reicht es an entsprechende Business Activities oder Business Entities weiter. Analoges gilt für den umgekehrten Weg vom Server zum Klienten. Durch Value Objects können im Gegensatz zu Business Entities auch komplexere Datenstrukturen ausgedrückt werden, um näher an den Anforderungen der Anwendung zu bleiben. Es ist dann jedoch ein Mapping notwendig, das die Strukturen an die Business Entities anpasst.

5.4 CA und MDA

Die CA bietet eine abstrakte Architektur, die an keine Plattform gebunden ist. Somit besteht die Möglichkeit, Anwendungen durch ein von der MDA gefordertes PIM auszudrücken. Sie stellt eine Terminologie für die Designsprache zur Verfügung, die sich in den Kern-Komponenten wie Business Object, Business Activity und gewissen Einschränkungen (Constraints) des Entwurfs ausdrückt. Ein PIM wird durch ein erweitertes UML Metamodell beschrieben – die Terminologie der CA legt fest, um welche (Sub-)Klassen das Metamodell erweitert wird. Die CA definiert im weitesten Sinne ein UML Profil [OMG04] ähnlich des in Kapitel 3.4 erwähnten Profils für verteilte Anwendungen (EDOC).

6 b+m Generator Framework

Das b+m Generator Framework ist ein template-gestützter Code-Generator der b+m Informatik AG, der sich an MDA orientiert bzw. den darauf aufbauenenden *Generative Development Process* (GDP) verwendet. Das Framework existiert als Open Source Projekt bei SourceForge.net [b+mGenFw], Binaries und Quellcode sind dort frei erhältlich und stehen unter der LGPL (GNU Lesser General Public License). Der Generator wird in der Diplomarbeit dazu verwendet, das PIM einer serverseitigen Architektur über metamodell-basierte Templates in Quellcode zu transformieren.

6.1 Überblick

Der b+m Generator ist keine „out-of-the-box"-Enwicklungsumgebung mit einem grafischen Frontend, Editoren und sonstigen Werkzeugen, sondern auf das Wesentliche reduziert: die Transformation von UML-Diagrammen mithilfe von Metamodellen in Quellcode. Für den Entwicklungsprozess (Modellierung, Implementierung, Build und Deployment) müssen weitere Werkzeuge hinzugezogen werden.

Generelle Funktionsweise des Generators

Als Input dient dem Generator ein nach XMI exportiertes Anwendungsdesign (das PIM), eine Reihe von Templates und eine in Java implementierte Version des Metamodells, die Klassen- und Zustandsdiagramme beschreibt. Ein Instanziator erzeugt aus den XMI Daten ein Objektgeflecht von Klassen des erweiterten Metamodells im Hauptspeicher. Das erweiterte Metamodell beschreibt das Anwendungsdesign und erlaubt den Zugriff und die Navigation innerhalb des Modells zur Laufzeit. Die Templates sind dynamisch an das Metamodell gebunden und expandieren daraus den Quellcode als Output.

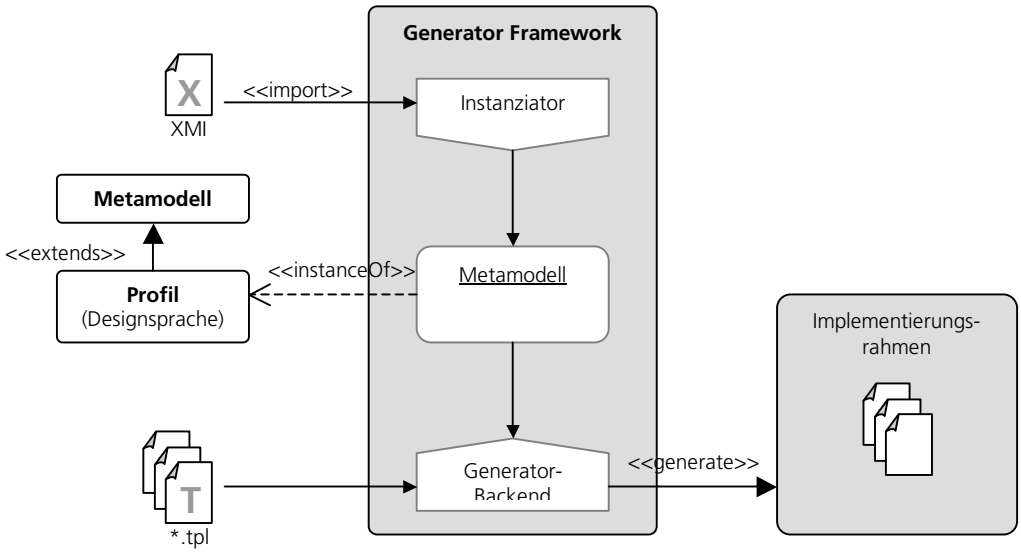

Abbildung 17: Funktionsweise des b+m Generator Frameworks

6.2 Erweiterbarkeit

Das b+m Generator Framework ist ein schlankes, offenes und sehr flexibles Framework. Einige Dinge wie das Mapping des XMI Imports oder das Mapping der Metamodell-Erweiterung lassen sich deklarativ in XML erledigen. Über einen Template-Mechanismus hat man detaillierte Kontrolle, was und wie generiert werden soll.

Zwei wichtige Erweiterungsmöglichkeiten sollen hier hervorgehoben werden:

XMI Import

UML Modellierungswerkzeuge interpretieren den XMI Standard z. T. recht unterschiedlich und exportieren Diagramme in verschiedener Struktur und Komplexität in das XMI Format. Daraus ergeben sich zwei Anforderungen an den Generator:

- es müssen unterschiedliche XMI Strukturen importiert werden können
- überflüssige Informationen sollten herausgefiltert werden können - nicht alle Details eines Exports sind für die Instanziierung des Metamodells notwendig.

Deshalb bietet der Generator eine Adapter Schnittstelle, über die man je nach verwendetem Werkzeug ein Mapping zwischen XMI Import und der internen Repräsentation festlegen kann. Das Mapping wird in XML ausgedrückt und ist für das jeweilige UML Werkzeug im Package `de.bmiag.genfw.instantiator.toolsupport.baseuml.<UML-Werkzeug>` des b+m Generator Frameworks zu finden.

Metamodell Erweiterung

Das Framework liefert in Java implementierte Versionen von Metamodellen für Klassen-, Zustands- und Aktivitätsdiagramme. Die Unterstützung weiterer Metamodelle ist möglich und erfordert lediglich deren Umsetzung in Java sowie das Formulieren entsprechender Instanziierungsvorschriften.

Um Architekturfamilien abbilden zu können, ist eine Erweiterung der Metamodell-Implementierung erforderlich. Das Mapping von Stereotypen im PIM auf die jeweilige Erweiterungsklasse kann über eine weitere Mapping-Datei metamapping.xml sehr flexibel angegeben werden.

6.3 Metamodell-Instanziierung

Die Instanziierung eines Metamodells zur Laufzeit wird durch sogenannte Instanziierungsvorschriften festgelegt, die das vorliegende Anwendungsdesign in XML als Datengrundlage nutzen. Die Vorschriften selbst sind in XML notiert.

6.4 Templates, XPand Scriptsprache

Templates werden in einer oder mehreren Textdateien mit der Endung .tpl definiert. Ein Template besteht aus Anweisungen der Scriptsprache XPand, die den variablen Anteil bilden und Bruchstücken einer Programmiersprache der Zielplattform (z. B. Java), die den statischen Anteil bilden. Die Textdatei selbst definiert den Namensraum von Templates.

Bei XPand handelt es sich um eine einfache, von b+m Informatik AG definierte Scriptsprache, über die man die variablen Anteile eines Templates durch das Anwendungsdesign generieren kann.

Die statischen Anteile werden direkt in Quellcode ausgegeben, während die Template Engine des Frameworks aus den XPand Anweisungen Quellcode erst zur Laufzeit erzeugt.

Eine XPand Anweisung ist eine in sich geschlossene Einheit, während statischer Code von XPand Anweisungen unterbrochen werden darf. XPand bietet die typischen Sprachkonstrukte wie bedingte Anweisungen, Schleifen und Operatoren. Eine XPand Anweisung wird mit Klammern «» dargestellt und kann sich über einen Geltungsbereich (Scope) erstrecken, der durch ein Anfangs- und Endschlüsselwort definiert ist. Die genaue Spezifikation mit allen Anweisungen ist unter [b+m02] zu finden.

```
«DEFINE <TemplateName> FOR <MetaClass>»some Code«ENDDEFINE»
```

XPand besitzt die besondere Eigenschaft, dass man in den Templates über das erweiterte Metamodell auf das Anwendungsdesign zugreifen kann. Der Zugriff wird durch die Template-Definition möglich. In der Template-Definition ordnet man ein Template der entsprechenden Klasse im Metamodell zu.

Innerhalb des Geltungsbereichs des Templates kann man nun auf Methoden der Klasse des Metamodells zugreifen. Für den Aufruf einer Methode wird lediglich ihr Name ohne Methoden-Klammern in Form einer XPand Anweisung notiert. Kaskadierende Methodenaufrufe werden wie in Java durch einen Punkt getrennt.

```
«DEFINE BusinessEntityClass FOR BusinessEntity»
      public abstract class «Name»BEBean implements EntityBean {
            ...
      }
«ENDDEFINE»
```

Listing 1: BusinessEntityClass-Template

Expansion

Unter Expansion versteht man das Umwandeln von XPand Anweisungen oder Templates in Quellcode. Dazu werden Anweisungen durch das Schlüsselwort `EXPAND` ergänzt oder die Anweisung `«EXPAND ...»` direkt notiert.

Der Zugriff auf das Metamodell über XPand erlaubt weitreichende Möglichkeiten wie etwa die Navigation durch das Metamodell: Wenn man beispielsweise von einer Klasse BusinessEntity des erweiterten Metamodells auf alle Klassen zugreifen möchte, die mit BusinessEntity über eine Assoziation verbunden sind, so kann man das so notieren:

```
«DEFINE BusinessEntityClass FOR BusinessEntity»
      «FOREACH AssociationEnd AS curAssoc EXPAND»
      public abstract «curAssoc.Opposite.Class.Name»BE
            «curAssoc.Opposite.Name.asGETTER»();
      «ENDFOREACH»
«ENDDEFINE»
```

Listing 2: Iteration über alle Assoziationsenden der BusinessEntity.

In der Schleife wird über alle Assoziationsenden iteriert und zur jeweiligen Klasse am anderen Ende der Assoziation navigiert. Der Name der Klasse wird dann für die Benamung einer getter Methode verwendet.

Template-Struktur

Die Reihenfolge der Expansion von Templates wird in einer speziell reservierten Template-Datei Root.tpl festgelegt. Mit ihr kann man verschiedene Einstiegspunkte für den Generator in das Metamodell definieren. Der tatsächlich ausgeführte Einstiegspunkt muss dem Generator als Kommandozeilenargument übergeben werden. Dies ermöglicht eine flexible Generierung von einzelnen Komponenten des Anwendungsdesigns.

***Beispiel*:** Im Anwendungsdesign gibt es eine Komponente, die mehrere Klassen enthält. Man kann nun das Root-Template für die Komponente definieren und weitere Templates für die Klassen. Im Root-Template expandiert man dann die Templates der Klassen. Die Komponente wird dem Generator als Einstiegspunkt angegeben.
Wenn man nun jedoch nicht die ganze Komponente generieren möchte, sondern lediglich eine der Klassen, so kann man das Root-Template für diese Klasse definieren und sie dem Generator als Einstiegspunkt angeben.

```
«REM Einstiegspunkt für Component»
«DEFINE Root FOR Component»
      «EXPAND Root FOREACH Class»
«ENDDEFINE»

«REM Einstiegspunkt für Class»
«DEFINE Root FOR Class»
      «EXPAND MyClassTemplate::MyClassTemplate»
«ENDDEFINE»
```

Listing 3: Template-Definitionen in Root.tpl.

In Kapitel 7.7 sieht man die konkrete Anwendung des Template-Mechanismus´.

6.5 Geschützte Bereiche

Ein wesentlicher Bestandteil des Generator Frameworks ist die Möglichkeit, innerhalb der generierten Artefakte geschützte Bereiche (Protected Regions) zu definieren. Geschützte Bereiche werden bei einer wiederholten Generierung nicht gelöscht oder überschrieben. In geschützten Bereichen können Entwickler manuelle Ergänzungen vornehmen. Dies ist notwendig, um Fachlogik oder Werte zu implementieren, die nicht aus dem Anwendungsdesign selbst generiert werden können.
Ein geschützter Bereich muss für den Entwickler als solcher gekennzeichnet und für den Generator eindeutig identifizierbar sein, ansonsten kann der Generator Bereiche nicht voneinander unterscheiden.

Die Template-Sprache XPand stellt dazu eine spezielle Anweisung für geschützte Bereiche zur Verfügung:

```
«PROTECT CSTART <String> CEND <String> ID <Id>»
      // TODO: enter your own code here...
«ENDPROTECT»
```

`<String>` ist in der Regel die Kommentar-Notation der Sprache der Zielplattform, bei Java beispielsweise „`//`“. `<Id>` ist die eindeutige Identifikation. Hier kann man entweder hartcodiert einen beliebigen eindeutigen Wert eintragen oder eine Methode des Metamodells aufrufen, die eine UID zurückgibt.

Im generierten Quellcode entsteht aus dieser Anweisung beispielsweise folgender Kommentar:

```
//PROTECTED REGION ID(#16a786:f8f4003962:-7f84#classBodyRegion) START
       // TODO: enter your own code here...
//PROTECTED REGION END
```

Der Entwickler darf natürlich unter keinen Umständen die Notation des geschützten Bereichs oder dessen UID verändern, ansonsten gehen bei einer erneuten Generierung alle manuellen Ergänzungen verloren.

6.6 Design Constraints

Design Constraints sind Vorschriften und Einschränkungen des Anwendungsdesigns, an die sich ein Modellierer halten muss. So kann man z. B. über ein Constraint vorschreiben, dass zwischen zwei bestimmten Klassen keine Beziehung existieren darf oder dass einer Komponente nur bestimmte Klassen zugeordnet werden dürfen. Design Constraints selbst werden in der Erweiterung des Metamodells implementiert und zur Laufzeit des Generators überprüft. Dazu sieht das Generator Framework eine Methode `CheckConstraints()` in der Klasse `de.bmiag.genfw.meta.Element` vor, die Bestandteil der Metamodell-Implementierung ist. Diese Methode kann durch entsprechende Subklassen überschrieben werden. So kann man in `CheckConstraints()` den Zustand des instanziierten Metamodells überprüfen und beispielsweise eine Ausnahme werfen, wenn eine Einschränkung im Anwendungsdesign nicht eingehalten worden ist. Der Einstiegspunkt für die Überprüfung der

Einschränkungen im Metamodell wird analog dem Root-Template über eine spezielle Template-Datei Check.tpl festgelegt, in dem lediglich die `CheckConstraints()` Methode aufgerufen wird.
Das Überprüfen von Design Constraints stellt eine mächtige Möglichkeit dar, nur aus PIMs zu generieren, die sich an eine vorgebene Architektur halten. Leider werden Design-Fehler erst zur Generierungszeit erkannt und nicht zur Modellierungszeit, was dem Modellierer einiges an Disziplin abverlangt.

6.7 Konfigurationsparameter

Der b+m Generator wird über den Aufruf der VM mit der Klasse `de.bmiag.genfw.Generator` über die Systemkonsole oder ein Ant Skript gestartet. Dabei sind für den Generator einige Angaben erforderlich, um die Ausführung des Generierungsprozesses zu konfigurieren. Die Angaben werden in Form von Kommandozeilenparametern an die Klasse übergeben.

`-classpath`: Der Klassenpfad enthält die Bibliotheken, die direkt vom Generator benötigt werden und im Release inbegriffen sind, sowie die Klassen der Metamodell-Erweiterung und evt. darin verwendete weitere Bibliotheken wie beispielsweise JDOM.

Die folgenden Parameter sind in der Form `<parameter-name>=<value>` anzugeben:

- `de.bmiag.genfw.ui.class`: Die Benutzeroberfläche, die verwendet werden soll. Für reinen Batchbetrieb ist hier `de.bmiag.genfw.ui.text.TextUI` anzugeben.
- `de.bmiag.genfw.instantiator.design`: Der Pfad der XMI Datei des Anwendungsdesigns.
- `de.bmiag.genfw.instantiator.xmlmap`: Der Pfad der XML Mapping Datei, die den XMI Export des UML Werkzeugs auf die interne Repräsentation abbildet.
- `de.bmiag.genfw.instantiator.tooladapter.class`: Die ToolAdapter Klasse, die dem verwendeten UML Werkzeug entspricht.
- `de.bmiag.genfw.xpand.path`: Der Verzeichnispfad, in dem sich die Templates befinden.
- `de.bmiag.genfw.instantiator.metamap`: Der Pfad der XML Mapping Datei, die die Stereotypen auf die erweiterten Klassen des Metamodells abbildet.
- `de.bmiag.genfw.baseuml.identifier.class`: Der vollqualifizierte Name der Indentifier Klasse des Metamodells. Hier wird der vorhandene Identifier durch die Erweiterung `de.softlab.mda.metamodel.Identifier` überschrieben.
- `de.bmiag.genfw.prresolver.path`: Der Pfad auf bereits generierte Dateien, die geschützte Code-Bereiche enthalten.
- `de.bmiag.genfw.filewriter.path`: Der Pfad, in dem die generierten Dateien gespeichert werden sollen.
- `de.bmiag.genfw.textui.check`: Schalter, der angibt, ob Design Constraints überprüft werden sollen (true l false).
- `de.bmiag.genfw.textui.select`: Die Auswahl eines Root-Templates, das den Ablauf der weiteren Generierung bestimmt. Hier wird BusinessComponent angegeben, um die Generierung der gesamten Komponente zu starten.

Es sind noch weitere Parameter für das Logging und Debugging möglich, sollen hier aber nicht explizit angeführt werden.

```
<java classname="de.bmiag.genfw.Generator" fork="true" >
      <classpath>
            <fileset dir="${lib}" includes="*.jar"/>
            <fileset dir="${lib}/jdom" includes="*.jar"/>
            <pathelement path="${metamodel}"/>
      </classpath>
      <arg value="de.bmiag.genfw.ui.class=de.bmiag.genfw.ui.text.TextUI"/>
      <arg value="de.bmiag.genfw.instantiator.design=design.xmi"/>
      <arg value="de.bmiag.genfw.instantiator.xmlmap=
            ${xmlmapPath}/poseidon20_xmi12_all.xml"/>
      <arg value="de.bmiag.genfw.instantiator.tooladapter.class=
            de.bmiag.genfw.instantiator.xml
            .toolsupport.baseuml.poseidon.PoseidonAdapter"/>
      <arg value="de.bmiag.genfw.xpand.path=templates"/>
      <arg value="de.bmiag.genfw.instantiator.metamap=metamapping.xml"/>
      <arg value="de.bmiag.genfw.baseuml.identifier.class=
            de.softlab.mda.metamodel.Identifier"/>
      <arg value="de.bmiag.genfw.prresolver.path=${generated}/backup"/>
      <arg value="de.bmiag.genfw.filewriter.path=${generated}/src"/>
      <arg value="de.bmiag.genfw.textui.check=false"/>
      <arg value="de.bmiag.genfw.textui.select=BusinessComponent"/>
</java>
```

Listing 4: Aufruf des Generators über eine Ant Task.

7 Umsetzung

Die Umsetzung gehört zum praktischen Teil der Diplomarbeit und beschreibt die Einrichtung der Laufzeitumgebung, die Auswahl der Werkzeuge und die Konfiguration des b+m Generator Frameworks. Die Erweiterung des Metamodells und die Generator-Templates werden in einem eigenen Unterkapitel ausführlich behandelt.

7.1 Laufzeitumgebung

Die Laufzeitumgebung ist für die Ausführung der Implementierung und deren Test notwendig. Sie orientiert sich im Wesentlichen an den Gegebenheiten der bei der BMW AG eingesetzten Umgebungen, so dass eine spätere Verwendung unter "realen" Bedingungen ohne große Änderungen möglich ist.

7.1.1 Application Server

Für die Ausführung der Anwendung wird der Weblogic Application Server (WAS) Version 8.1 von BEA Systems Inc. verwendet. Der WAS ist eine kommerzielle, professionelle und praxiserprobte Application Server Plattform, die zahlreiche Dienste für verteilte Anwendungen auf Basis der J2EE Plattform zur Verfügung stellt. Ein wichtiger Service ist die Bereitstellung des EJB Containers, in dem EJB Komponenten ausgeführt werden können. Der WAS ist über ein Web Frontend komfortabel konfigurierbar, mehrere kleine Hilfsmittel wie etwa ein Konfigurator für EJB Komponenten sind im Lieferumfang enthalten. Unter der Motorhaube verwendet der Server für die Ausführung von Java Bytecode eine von BEA eigens entwickelte virtuelle Machine (JRockit), die speziell an serverseitige Anwendungen auf Intel-Plattformen angepasst ist, um eine optimale Performance zu ermöglichen.
Der WAS wird bei der BMW AG von vielen Client/Server Anwendungen eingesetzt.

Die Dokumentation auf der Webseite [Weblogic] von BEA ist sehr umfangreich und hilfreich. Im praktischen Einsatz haben sich die ausführlichen Fehlermeldungen als sehr positiv herausgestellt, da sie beim Debugging von großem Nutzen waren.
Da der WAS sehr ressourcenfressend ist, kann es bei einer lokalen Installation auf einem schwachbrüstigen PC-System zu erheblichen Performance-Einbußen kommen, deshalb ist ein Hauptspeicher von mindestens 512 MB unbedingt empfehlenswert.

Konfiguration

Der lokale Installationsvorgang des WAS unter Windows 2000 soll hier nicht näher erläutert werden, da man dort mithilfe eines Wizards durchgeleitet wird. Interessant ist die anschließende Konfiguration des Servers. Diese basiert auf einer Konfigurationsdatei config.xml, die jedoch nicht manuell erstellt werden sollte, sondern mithilfe des mitgelieferten Configuration Wizards.

Der Wizard leitet durch die folgenden Wizard-Seiten:

1. *Create a Configuration or Add to an existing Configuration:* Hier wird der Radio-Button „create a new Weblogic configuration" ausgewählt
2. *Select a Configuration Template*: Aus der Baumansicht wird die „Basic Weblogic Server Domain" ausgewählt

3. *Choose Express or Custom Configuration*: "Custom"
4. *Administration Server Configuration*: Auf dieser Seite wird der logische Name der Server-Instanz, dessen IP-Adresse und Port angegeben (mda-server, localhost:7001).
5. *Multiple Servers, Clusters, and Machine Options*: Diese Seite wird übergangen.
6. *Database (JDBC) Options*: Die Option „Yes" wird ausgewählt.
7. *JDBC Connection Pool Configuration*: Hier wird ein neuer Connection-Pool unter dem Namen „mda-pool" und dem JDBC-Treiber „COM.ibm.db2.jdbc.app.DB2-Driver" zur Konfiguration hinzugefügt. Der Treiber ist aus einer Klappliste wählbar. Für die Identifikation der Datenbank wird die URL „jdbc:db2:MDATA" angegeben. Etwas irreführend ist das Properties-Eingabefeld. Dort wird der Benutzername für die Datenbank erwartet, muss aber in der Form „user=Benutzername" in das Feld eingetragen werden. Als Benutzername und Passwort wurde hier der Windows 2000 Log-in benutzt.
8. *JDBC Multipool Configuration*: Diese Seite wird übergangen.
9. *Configure JDBC Data Source*: Auf dieser Seite wird der logische Name der Datenquelle („mda-datasource") auf den JNDI Namen („mda-datasource") und auf den Connection Pool („mda-pool") gemapped. Dieser Name wird später im technischen Deployment Deskriptor weblogic-cmp-rdbms-jar.xml für jede EntityBean angegeben.
10. *JDBC Database Configuration*: Hier ist es möglich, den Zugriff auf die Datenbank zu testen. Das hat jedoch aus unerfindlichen Gründen noch für keinen Konfigurationsfall funktioniert, obwohl die Datenbank mit den selben Konfigurationsdaten über einen herkömmlichen Klienten ansprechbar ist.
11. *Messaging (JMS) Options*: Diese Seite wird übergangen.
12. *Targeting Options*: Diese Seite wird übergangen.
13. *Advanced Security Options*: Diese Seite wird übergangen.
14. *Configure Username and Password*: Hier kann man den Benutzernamen und das Passwort angeben, mit dem man sich nach der Konfiguration in die Administrationskonsole des WAS einloggen kann.
15. *Windows Options*: Die Default-Einstellungen können übernommen werden. Damit wird ein Shortcut-Icon zum Start des Servers im Start-Menü angelegt.
16. *Build Start Menu Entries*: Hier kann die Batch-Datei angegeben werden, die zum Start des Servers ausgeführt werden muss.
17. *Weblogic Configuration Environment*: Hier kann man zwischen Produktions- oder Development Modus und dem verwendeten JDK auswählen. Die in der Diplomarbeit verwendete Konfiguration nutzte das separat installierte JDK 1.4.2 von Sun.
18. *Create Weblogic Configuration*: Auf dieser Seite wird der Pfad und der Name der Domäne angegeben, für die die Konfiguration gelten soll. Durch Aktivieren des Create Buttons wird die Konfiguration in der Datei config.xml persistent gemacht und ein Start-Skript (startWeblogic.cmd) angelegt.

Nach dem Konfigurationsvorgang ist es möglicherweise notwendig, das Start-Script anzupassen, um beispielsweise zusätzliche Angaben im Klassenpfad zu machen oder den Server im Debug-Modus zu starten.

7.1.2 Datenbank

Für die Persistenzschicht wurde das relationale Datenbanksystem DB2 Version 7 der IBM Corporation [DB2] verwendet. DB2 ist auch bei BMW produktiv im Einsatz, unter anderem ist der Datenbestand von IVS-R mit dieser Datenbank gespeichert. Wie schon der WAS ist DB2

ein professionelles System für industrielle Anwendungen, vergleichbar mit Oracle Datenbanken. Bestandteil der Installation ist auch ein Client Werkzeug, über das SQL-Anweisungen an die Datenbank abgesetzt werden können.

Konfiguration

Auf der Datenbank wurden die Tabellenstrukturen der IVS-R Stammdatenverwaltung eingerichtet, die als Grundlage für die Persistenzschicht des Projektbeispiels Händler-Stammdatenverwaltung (vgl. Kapitel 8.1.5) dienen. Die Erstellung der Tabellen und Views erfolgte über ein Datenbank-Skript, das die entsprechenden SQL-Anweisungen bereitstellte.
Die DB2 Installation bietet einige Werkzeuge zur Administration und Konfiguration der Datenbank. Über die „Steuerzentrale" kann man dem Datenbanksystem eine neue Datenbank hinzufügen, die für die Stammdatentabellen „MDATA" genannt wurde. Für die Tabellen wurde das Schema „OIT" festgelegt. Informationen wie vorhandene Tabellen und Sichten einer erstellten Datenbank erhält man aus der Baumansicht der Steuerzentrale. SQL-Skripte oder einzelne SQL Anweisungen lassen sich über die „Befehlszentrale" an die Datenbank absetzen.

7.1.3 Test Client

Für das Testen der an den Application Server ausgelieferten Anwendung kommt ein kleiner Test Client in Form von [JUnit] Test-Cases zum Einsatz, der über die virtuelle Maschine der Eclipse Plattform gestartet wird. Die Test-Case Klasse wird mithilfe des b+m Generators aus dem PIM generiert und enthält Testmethoden, die die Anwendungsfälle abbilden, jedoch noch nicht ausimplementiert sind. Der Zugriff des Clients erfolgt über einen JNDI-Lookup der Home Factory der Session Facade.

7.2 Werkzeuge und Bibliotheken

Für den gesamten Entwicklungsprozess (Analyse, Design, Implementierung, Generierung, Build und Deployment) wurden verschiedene, grösstenteils frei erhältliche Werkzeuge und Bibliotheken verwendet. Manche sind Status Quo im professionellen Umfeld (z. B. Ant, JUnit) und Mittel der Wahl, während andere zunächst gesucht und evaluiert werden mussten (z. B. Poseidon, b+m Generator).

Eclipse IDE

Als integrierte Entwicklungsumgebung für die Projektverwaltung und Implementierung von Java-Code wurde die Eclipse Plattform Version 2.1.2 [Eclipse] verwendet.
Die Eclipse Plattform ist ein Open Source Projekt von IBM, deren fundamentale Architektur aus einem ausgeklügelten Erweiterungsmechanismus besteht, über den sich IDEs aller Art (J2EE, C++) und sonstige Funktionalität in Form von Plugins einbinden lassen. Eclipse stellt einen integrierten Java Editor und Debugger, Ant Unterstützung und eine JUnit View zur Verfügung. Der Java Editor bietet eine Vielzahl an Code Refactoring Funktionalität und erleichtert dadurch die Programmierung erheblich.
Die Oberfläche von Eclipse besteht aus der sogenannten Workbench, in die Views und Editoren integriert sind.

Ant

Für den Workflow von der Generierung über den Build bis zum Deployment wurde [Ant] verwendet, ein Open Source Projekt von Jakarta (jakarta.apache.org). Ant ist vergleichbar mit

Make unter Unix, Konfigurationen werden jedoch in einem XML-Dialekt ausgedrückt, was sie leichter lesbar macht und besser strukturiert.
Einzelne Prozesse eines Workflows können als sogenanntes „Targets“ in der XML-Datei definiert werden. Targets rufen eine oder mehrere Ant Tasks auf wie beispielsweise die Kompilierung von Java Dateien oder die Ausführung eines Windows Prozesses. Ant ist in Eclipse als Plugin integriert und stellt eine spezielle View auf die XML Konfigurationsdatei zur Verfügung, in der die einzelnen Targets sichtbar sind.

b+m Generator Framework

Das b+m Generator Framework wurde bereits in Kapitel 6 näher erläutert. In der Diplomarbeit wurde die Version 2.x verwendet. Das Framework ist zu Beginn der Diplomarbeit von der b+m Informatik AG unter die LGPL gestellt und der Quellcode über SourceForge [b+mGenFw] veröffentlicht worden. Die kompilierte Version besteht aus zwei JAR Archiven. Das eine enthält die Java-Implementierung des Metamodells für Klassen und Zustandsdiagramme, das andere die Implementierung des Framworks selbst.
Das Projekt bei SourceForge stellt über ein CVS eine Referenzdokumentation und ein Diskussions-Forum zur Verfügung, das während der Diplomarbeit wertvolle Tips geliefert hat. Weitere Dokumente sind auch auf der Web-Seite der b+m Informatik AG [b+mGenFw] erhältlich.

Visual Sourcesafe

Für das Backup und die Versionierung der Eclipse-Projekte wurde Visual Sourcesafe (VSS) von Microsoft [VSS] eingesetzt. VSS ist vergleichbar mit CVS, ausgecheckte Dateien sind jedoch gesperrt, so dass immer nur eine Person Schreibrechte hat. Der Zugriff erfolgt über eine VSS Client Software, die man als Plugin in Eclipse integrieren kann. Neben dem Nutzen als Backup-System hat VSS den Vorteil, dass auch andere Mitarbeiter aus Eclipse Zugriff auf die Quelldateien haben.

Poseidon UML

Für die Modellierung des PIM wurde das UML CASE Tool Poseidon 2.1.1 Community Edition von Gentleware AG [Poseidon] eingesetzt. Das Werkzeug erlaubt das Zeichnen der gängigen UML Diagramme und besitzt eine XMI Export Schnittstelle, über die die Diagramme als XML Datei abgespeichert werden können. Auschlaggebend für die Wahl dieses Werkzeugs war die kostenlose Verfügbarkeit, die Kompatibilität des XMI Formats mit dem bereits vorhandenen Mapping (vgl. Kapitel 6.2) des b+m Generator Frameworks und die Möglichkeit, Modellierungselemente innerhalb von Komponenten zu platzieren. Negativ aufgefallen ist die Useability der Anwendung, insbesondere beim Zeichnen von Modellierungselementen.

JUnit

[JUnit] ist ein Framework zum Testen von Anwendungen über Testfälle (TestCases) und ist (wie schon Ant) als Plugin in Eclipse integriert. Ein Testfall kann als eigene Run-Task in Eclipse gestartet werden und visualisiert das Ergebnis des Tests in einer speziellen View. Mehrere JUnit TestCases dienten generierten EJB Anwendungen als Client.

Jalopy Code Formatter

[Jalopy] ist ein Open Source Werkzeug zur Formatierung von Java Code. Es ist als eigenständige Anwendung oder als Eclipse-Plugin ausführbar. Jalopy verfügt über eine Vielzahl von Einstellungen, mit denen die Art der Formatierung verändert werden kann. Die Einstellun-

gen werden in einer XML Datei abgepeichert und können dort entweder manuell oder über eine Swing GUI modifiziert werden.
Jalopy wurde dazu verwendet, die generierten Artefakte (Java Klassen) in eine besser lesbare Form zu bringen, da der b+m Generator zum Stand der Diplomarbeit äussert schlecht formatierten Code erzeugt.

7.3 Projekt Struktur

Die Artefakte des praktischen Teils der Diplomarbeit sind im Visual Sourcesafe Repository als Projekt unter dem Namen „MDA“ angelegt. Das MDA Projekt teilt sich in weitere Subprojekte auf, für die jeweils eigene Unterverzeichnisse vorgesehen sind:

MDA-Generator Projekt

Die Erweiterung des b+m Generator Frameworks ist ein eigenständiges Eclipse-Projekt und im Verzeichnis „MDA-Generator“ im Repository abgespeichert. Projekte, die eine Anwendungsarchitektur mithilfe des b+m Generator Frameworks generieren möchten, müssen dieses Projekt verwenden. Wie das im einzelnen abläuft, wird in Kapitel 8 detaillierter beschrieben. Das Projekt enthält im Wesentlichen die Templates und die Erweiterung des Metamodells für Klassendiagramme, sowie etliche Bibliotheken, die für die Ausführung des gesamten Build- und Deployment Prozesses erforderlich sind.

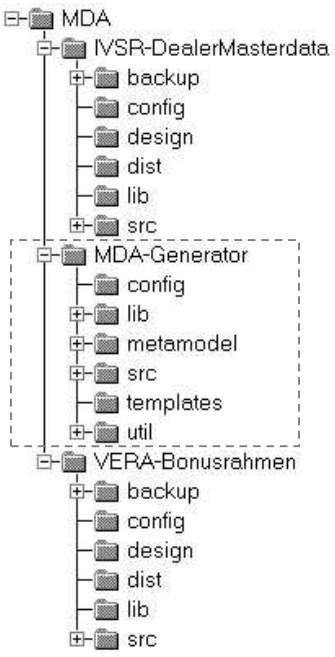

Abbildung 18: Die Verzeichnisstruktur der Projekte im VSS

Die einzelnen Ordner und Dateien haben folgende Bedeutung:

/config: Enthält die Konfigurationsdateien für das Jalopy Code-Formatter Utility, das Mapping des Metamodells und die DTD der O/R Mapping-Datei map.xml

/lib: Enthält sämtliche Bibliotheken und Frameworks als JAR oder ZIP Archive, die für die Ausführung des Generierungs-, Build- und Deployment-Prozesses notwendig sind.
Zu nennen sind Jalopy Code Formatter, b+m Generator Framework, XML Parser, Metamodel Implementierung usw.

/metamodel: Enthält die Java-Klassen der Erweiterung des Metamodells für Klassendiagramme. Der Ordner unterteilt sich in den Unterordner /src, der den Quellcode enthält und /bin, in dem sich die kompilierten Class Dateien befinden. Die Package-Struktur der Implementierung ist `de.softlab.metamodel`.

/src: Enthält zum einen die Implementierung des projektunabhängigen Kerns der Anwendungsfamilie (Basisklassen, Exceptions, Factories, Utilities) und zum anderen die DTDs der Deployment Deskriptoren, sowie die Manifest-Datei, die für die Paketierung der JAR Archive verwendet werden.

/templates: Enthält die Generator-Template Dateien (*.tpl), die für die Generierung einer Anwendung nach den Vorgaben der BMW CA erforderlich sind.

/util: Enthält eigene Implementierungen von Utilities, die vom Build Prozess verwendet werden. Momentan ist dort das IDGenerator Utility (vgl. Kapitel 7.8) zu finden.

Anwendungs-Projekte (VERA-BonusRahmen, IVSR-DealerMasterdata)

Für Projekte, die mithilfe des Generators einen Architekturrahmen generieren wollen, sollte jeweils ein eigenständiges Eclipse-Projekt angelegt werden. Die allgemeine Ordner-Struktur ist folgendermaßen organisiert:

/config: Enthält die O/R Mapping Datei map.xml, sowie die Konfigurationsdatei config.properties.

/design: Enthält den XMI Export des Anwendungsdesigns als XML Datei. Hier können beispielsweise auch weitere Dateien des UML Werkzeugs gespeichert sein.

/lib: Enthält die Bibliotheken und Frameworks als JAR oder ZIP Archive, die mit dem EAR Archiv ausgeliefert werden müssen.

Die genannten Verzeichnisse sind zunächst die Basis für ein neues Projekt. Nach der erstmaligen Ausführung des Generierungs-, Build- und Deployment-Prozesses werden weitere Verzeichnisse automatisch angelegt:

/backup: Der backup Ordner enthält die generierten Dateien, die beim letzten Generierungsprozess erzeugt wurden. Die Dateien dienen dem Generator als Grundlage für das Ermitteln von geschützten Code-Bereichen, bevor neu generiert wird. Nach der Generierung

werden die neuen Dateien in das /backup Verzeichnis kopiert. Ginge beim Generierungsprozess etwas schief, hätte man immer noch ein aktuelle Version im /backup Ordner.

/bin: Enthält die kompilierten Klassen und Deployment Deskriptoren der Anwendung.

/dist: Enthält die paketierten Komponenten der Anwendung wie EAR und JAR Archive

/src: Enthält die mithilfe des b+m Generators erzeugten Artefakte aus dem Anwendungsdesign

7.4 Konfiguration des Transformationsprozesses

Der grösste Teil an Informationen, der für die Generierung von Quellcode benötigt wird, stammt aus dem PIM und den darin enthaltenen Modellierungselementen. Rein technische Informationen haben im PIM jedoch nichts zu suchen, sind aber trotzdem für eine erfolgreiche und möglichst vollständige Generierung notwendig.
Nun gibt es zwar die Möglichkeit, Informationen hartcodiert in Templates aufzubereiten und zur Verfügung zu stellen; in manchen Fällen scheitert dieser Ansatz jedoch an den eingeschränkten Möglichkeiten der Script-Sprache. Es ist beispielsweise nicht ohne weiteres möglich, mit einer XPand-Anweisung Daten aus anderen Dateien in ein Template einfließen zu lassen.
Aus diesem Grund wird die Erweiterung des Metamodells genutzt, um Logik für die Aufbereitung von externen Datenquellen bereit zu stellen. Da die konkrete Implementierung des Metamodells in Java implementiert ist, können alle Mechanismen und Möglichkeiten einer ausgereiften Programmiersprache genutzt werden.

Welche externen Informationen werden benötigt?

- Mapping-Informationen: Um den technischen Deployment Deskriptor vollständig generieren zu können, werden die technischen Namen der Tabellen und Spalten der Datenbank benötigt, die den Entities und deren Attribute zugeordnet sind. Assoziationen zwischen BusinessEntities im PIM müssen Datenbankrelationen zugeordnet werden. Hier sind Informationen über die Fremd- und Primärschlüssel-Spalten der Datenbanktabellen notwendig.
- Konfigurations-Informationen: Pfade und Packagenamen für den Output der generierten Artefakte müssen flexibel konfigurierbar sein. Weitere Konfigurationsoptionen wie die Generierung von Local oder Remote Interfaces sollten über Schalter einstellbar sein
-

Daraus ergibt sich die Anforderung, diese Informationen in externen Dateien zur Verfügung zu stellen, die über einen bestimmten Mechanismus von der Instanz des Metamodell eingelesen werden können.

- Mapping Informationen lassen sich am besten in XML ausdrücken und sind infolgedessen in der Datei map.xml gespeichert. Der XML-Dialekt orientiert sich stark an dem des Weblogic Deployment Deskriptors. Im Grunde genommen handelt es sich bei map.xml um einen vereinfachten Deployment-Deskriptor
- Konfigurationsinformationen werden in einer Properties-Datei config.properties gespeichert

Mapper

Die Implementierung des XML-Parsing der Datei map.xml und des Lookups für das Mapping ist in der Singleton Klasse `de.softlab.metamodel.Mapper` zu finden.

Der Einlesevorgang der Datei map.xml wird durch die Methode `map()` der Metamodell Klasse BusinessComponent angetriggert (genaugenommen muss `map()` aus einem Template heraus aufgerufen werden). `map()` erzeugt ein Mapper-Objekt und übergibt alle Instanzen von Klassen des Metamodells, die abgebildet werden müssen (BusinessEntity, Field, BusinessEntityRelation). Mapper ermittelt über den Objektnamen den entsprechenden technischen Namen und speichert ihn in demselben Objekt ab.

In den Templates kann anschließend über entsprechende Methoden auf die technischen Namen zugegriffen werden.

Beispiel: Der Mapper ordnet dem Objekt GroupDealer, das eine Instanz von BusinessEntity ist, über die Methode `BusinessEntity.setTableName()` den technischen Namen „OIT.VOID02“ der korrespondierenen Tabelle zu. In einem Template kann dann über «`BusinessEntity.getTableName`» darauf zugegriffen werden.

Etwas aufwändiger ist das Mapping von Datenbank-Relationen auf Assoziationen des PIM. Hier muss die Instanz des Metamodells (Objektmodell) um weitere Objekte (ForeignKey) ergänzt werden. Das Mapping wird in Kapitel 7.6.4 näher erläutert.

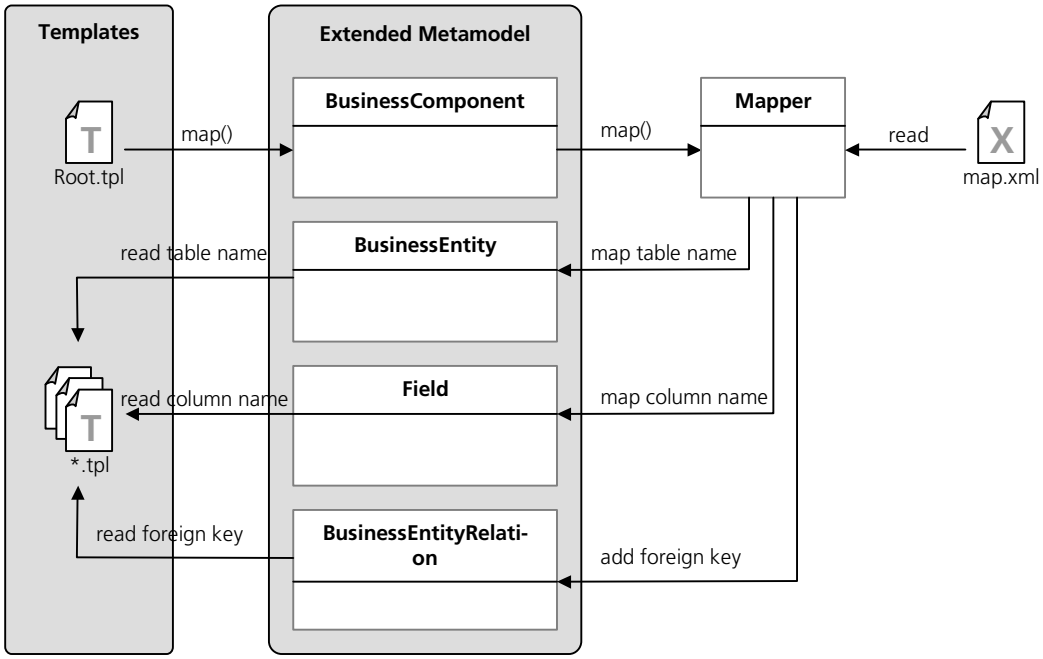

Abbildung 19: Funktionsweise des Mappers

PropertyProvider

Das Einlesen der Datei config.properties ist dagegen denkbar simpel: Die Singleton Klasse `de.softlab.metamodel.PropertyProvider` verwaltet die Properties-Datei und stellt eine Methode `getConfigProperty(String name)` zur Verfügung, über die aus dem Metamodell heraus Konfigurations-Einträge ausgelesen werden können.

Bespiel: Der allgemeine Package Name der generierten Artefakte soll aus der Properties Datei ausgelesen werden. Dazu ruft man in einem Template die Methode «`BusinessComponent.getCommonPackage`» `auf`. `BusinessComponent.getCommonPackage()` delegiert

den Aufruf an den PropertyProvider und liefert über einen entsprechenden Schlüssel den Package-Namen zurück.

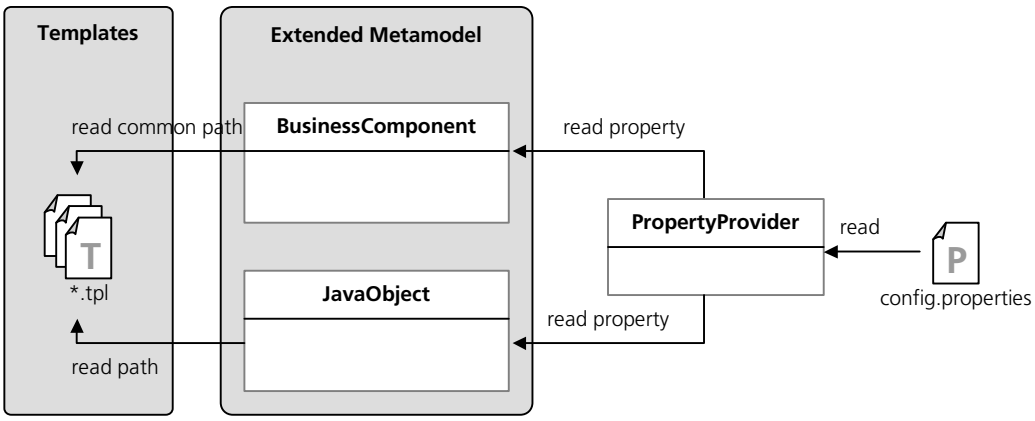

Abbildung 20: Die Funktionsweise des PropertyProviders

Die Beschreibung des Formats beider Dateien ist im Anhang B zu finden.

7.5 Metamodell-Erweiterung

Das UML Metamodell für UML Diagramme ist einer der Kernbestandteile eines MDA konformen Generators wie dem den b+m Generator. Die OMG sieht vor, dass man Metamodelle erweitern und anpassen darf, um sie für den spezifischen Einsatz in einer Problemdomäne verwenden zu können.

Warum muss man das Metamodell erweitern?

MDA Generatoren nutzen das Metamodell, um beim Transformationsprozess Informationen über das PIM zu erfahren und damit die Ausgabe des Quellcodes zu beeinflussen. Das Standard-Metamodell würde nun nichts darüber aussagen, für welche Architekturfamilie generiert werden soll, weil es keine spezifische Semantik enthält. Es wäre also nicht bekannt, ob nun die Architektur für ein verteiltes System oder ein Echtzeitsystem generiert werden soll. Dieses Problem löst man, indem man Klassen des Metamodells erweitert und deren Namen als Bezeichner für Stereotypen des PIM verwendet. In den Subklassen kann zusätzliche Logik wie etwa das Überprüfen von Design Constraints oder der Zugriff auf externe Daten untergebracht werden. Diese Funktionalität kann der Generator nutzen, indem er eine Instanz des Metamodells bildet. Es stehen über die Stereotypen Subklassen zur Verfügung, die über das Metamodell selbst oder in den Templates abgefragt werden können.

Welche Klassen des Metamodells muss man erweitern und wie benennen?

Hier kommen auf den ersten Blick die UML-Profile der OMG zum Einsatz. In einem UML-Profil wird das Metamodell bereits um die Semantik für die Zielplattform erweitert, d. h. es existiert innerhalb der Spezifikation ein Diagramm, das das erweiterte Modell abbildet. Für die Diplomarbeit wurde jedoch kein Profil der OMG verwendet, sondern die Spezifikation der BMW Component Architecture (CA). Sie ist zwar nicht als UML Profil ausgewiesen, hat aber beispielsweise Ähnlichkeiten mit dem EDOC Profil.

Nicht alle Erweiterungen sind durch die CA festgelegt, manche ergeben sich durch technische Notwendigkeiten. Man sollte aber darauf achten, dass die Erweiterungen nicht technisch benamt werden, wenn sie als Stereotypen im PIM Verwendung finden sollen.

Folgende Kernkomponenten der CA sind für die Erweiterung des Metamodells von Bedeutung

- Business Component
- Business Object
- Business Facade
- Business Activity
- Business Entity

Um das Metamodell erweitern zu können, muss eine konkrete Implementierung in einer Programmiersprache vorliegen. Das b+m Generator Framework enthält solch eine Implementierung in der Programmiersprache Java. Sie orientiert sich an der OMG Spezifikation zu UML [OMG01], wurde aber vereinfacht, da nicht alle Modellierungselemente für den praktischen Einsatz notwendig sind.
Das b+m Metamodell besteht aus insgesamt vier Packages:

- `de.bmiag.genfw.meta`: Enthält die Basistypen aller Metamodell-Klassen und stellt das Typsystem des Generator Frameworks dar.
- `de.bmiag.genfw.core`: Enthält die Klassen des UML Metamodell Kerns.
- `de.bmiag.genfw.classifier`: Enthält die Elemente, die in Klassendiagrammen verwendet werden.
- `de.bmiag.genfw.state`: Enthält die Elemente, die in Zustands- oder Activity-Diagrammen verwendet werden.

Erweiterung für Stereotypen

BusinessObject erweitert `de.softlab.mda.metamodel.JavaObject` und ist die gemeinsame Superklasse von BusinessFacade, BusinessActivity und BusinessEntity. Alle BusinessObjects sind Bestandteil von BusinessComponent.

BusinessComponent erweitert `de.bmiag.genfw.meta.classifier.Component` im Metamodell. Die BusinessComponent enthält alle BusinessObjects, die für die Ausführung der Komponente notwendig sind. Sie ist der Einstiegspunkt in das Metamodell bei der Instanziierung durch den Generator.
BusinessComponent stellt einige Hilfsfunktionen zur Verfügung, die in den Templates aufgerufen werden können:

- `CommonPackageName()`: Gibt den Namen des allgemeinen Package Teils der generierten Anwendung zurück z. B. „de.softlab.ivsr“.
- `CommonExceptionPackage()` : Gibt den Namen des Exceptions Package Pfads zurück, z. B. „de.softlab.mda.common.exception“.
- `ServerBasePackage()`: Gibt den Namen des Server Basis Packages zurück, z. B. „de/softlab/mda/server/base“.
- `ServerUtilPackage()`:Gibt den Namen des Server Utility Packages zurück, z. B. „de/softlab/mda/server/util“.
- `CommonPathName()`: Gibt den Namen des allgemeinen Pfades der generierten Anwendung zurück z. B. „de/softlab/ivsr“.

- `isLocal()`: Ein Schalter, der angibt, ob die EJB Komponenten mit Local oder Remote Interfaces generiert werden sollen.
- `map()`: Started das Mapping von Attributs- und Entity-Namen des PIM auf die technischen Namen der Datenbank (Tabellen, Spalten).

BusinessFacade erweitert `de.softlab.mda.metamodel.BusinessObject` und repräsentiert die Schnittstelle der Komponente nach aussen. Sie gibt über die Methode `BusinessActivity()` die BusinessActivity-Objekte zurück, die durch eine Dependency-Beziehung an sie gebunden sind.

BusinessActivty erweitert `de.softlab.mda.metamodel.BusinessObject` und implementiert die Fachlogik, die zur Ausführung einer fachlichen Operation notwendig ist.

BusinessEntity erweitert `de.softlab.mda.metamodel.BusinessObject` und repräsentiert die fachlichen Daten. Sie kann Attribute vom Typ Field und Key und Methoden vom Typ Finder enthalten.
BusinessEntity stellt einige Hilfsfunktionen zur Verfügung, die in den Templates aufgerufen werden können:

- `setTableName()`: Speichert den technischen Namen der Datenbanktabelle, die der Instanz der Metamodell-Klasse zugeordnet ist. Beispiel: Die Klasse GroupDealer im PIM ist eine BusinessEntity und entspricht der Datenbanktabelle `"OIT.VOID02"`.
- `getTableName()`: Gibt den Namen der technischen Datenbanktabelle zurück, die der Instanz der Metamodell-Klasse zugeordnet ist.
-

BusinessEntityRelation erweitert `de.bmiag.genfw.meta.classifier.Association` im Metamodell und bildet eine fachliche Assoziationsbeziehung auf eine technische Relation der Datenbank ab.

Field erweitert `de.bmiag.genfw.meta.classifier.Attribute` im Metamodell. Field ist durch eine technische Notwendigkeit entstanden, die in Kapitel 7.6.4 näher erläutert wird.
BusinessEntity stellt einige Hilfsfunktionen zur Verfügung, die in den Templates aufgerufen werden können:

- `setRdbmsColumnName()`: Speichert den technischen Namen der Spalte aus der Datenbanktabelle, die dem Attribut im PIM zugeordnet ist, z. B. Das Attribut „groupDealerID“ der Klasse GroupDealer im PIM entspricht der Spalte „NV_WS_ORDERER_ID“ der Tabelle „OIT.VOID02“ in der Datenbank.
- `getRdbmsColumnName()`: Gibt den technischen Namen der Spalte in der Datenbanktabelle zurück, die dem Attribut im PIM zugeordnet ist

Key erweitert `de.softlab.mda.metamodel.Field` und repräsentiert den eindeutigen Schlüssel einer BusinessEntity.

Finder erweitert `de.bmiag.genfw.meta.classifier.Operation` und ermöglicht das Finden von verteilten BusinessEntities über Primary Key Attribute. Finder entspricht der Finder Semantik bei EJB EntityBeans.

Technische Erweiterungen

Folgende Klassen werden nicht für die Auszeichnung verwendet. Sie haben rein technischen Charakter:

ForeignKey erweitert `de.softlab.mda.metamodel.Field` und repräsentiert den Fremdschlüssel, der sich auf eine BusinessEntity bezieht. Ein ForeignKey speichert einen Key, so dass eine Relation zwischen zwei Datenbanktabellen abgebildet werden kann.

JavaObject erweitert `de.bmiag.genfw.meta.classifier.Class` im Metamodell. JavaObject soll Java-spezifisches Verhalten kapseln. Für eine .NET Lösung würde man diese Klasse ersetzen.
JavaObject stellt einige Hilfsfunktionen zur Verfügung, die in den Templates aufgerufen werden können:

- `count()`: Ein einfacher Zähler, der eine eine um eins inkrementierte Zahl zurückgibt.
- `resetCounter()`: Setzt den Zähler auf den Ausgangswert zurück
- `PackageName()`: Gibt den Package Namen zurück, in dem das Generat dieser Metamodell-Klasse gespeichert werden soll
- `getUid()` und `setUid()`: Erlaubt das Speichern einer UID, die ein JavaObject eindeutig identifiziert

Identifier erweitert `de.bmiag.genfw.meta.core.Identifier` im Metamodell und hat rein technischen Charakter. Die Erweiterung der Klasse Identifier kann dem Generator als Kommandozeilenparameter angegeben werden, um die Default-Implementierung zu ersetzen. Identifier bietet verschiedene Hilfsfunktionen, um Namen von Instanzen der Klassen des Metamodells in den Templates in spezifischer Weise zu formatieren, beispielsweise

- `asVAR()`: Der Klassenname wird als Variablenname zurückgegeben, z. B. GroupDealer -> groupDealer
- `asJavaType()`: Ein im PIM technisch unabhängiger Typ wird als Java Typ gerendert, z. B. `String` -> `java.lang.String`
- `asPACKAGE()`: Der Klassenname wird als Packagename gerendert, z. B. GroupDealer -> groupdealer
- …

Operation erweitert `de.bmiag.genfw.meta.classifier.Operation` im Metamodell und hat rein technischen Charakter:

- `getUid()` und `setUid()`: Erlaubt das Speichern einer UID zu eindeutigen Identifikation der Operation/Methode.

Utility-Funktionen in der Metamodell-Erweiterung

Da nicht die gesamte Funktionalität durch die Template-Sprache imlementiert werden kann und auch Konfigurationsdaten von aussen in das Metamodell eingeführt werden müssen, wurden zusätzlich einige Hilfsfunktionen in das Metamodell implementiert (vgl. Kapitel 7.7.11).

Package-Struktur

Die implementierten Klassen der Metamodel-Erweiterung sind Bestandteil des MDA-Generator Projekts und im Unterverzeichnis /metamodel zu finden. Die Package-Struktur ist `de.softlab.mda.metamodel`.

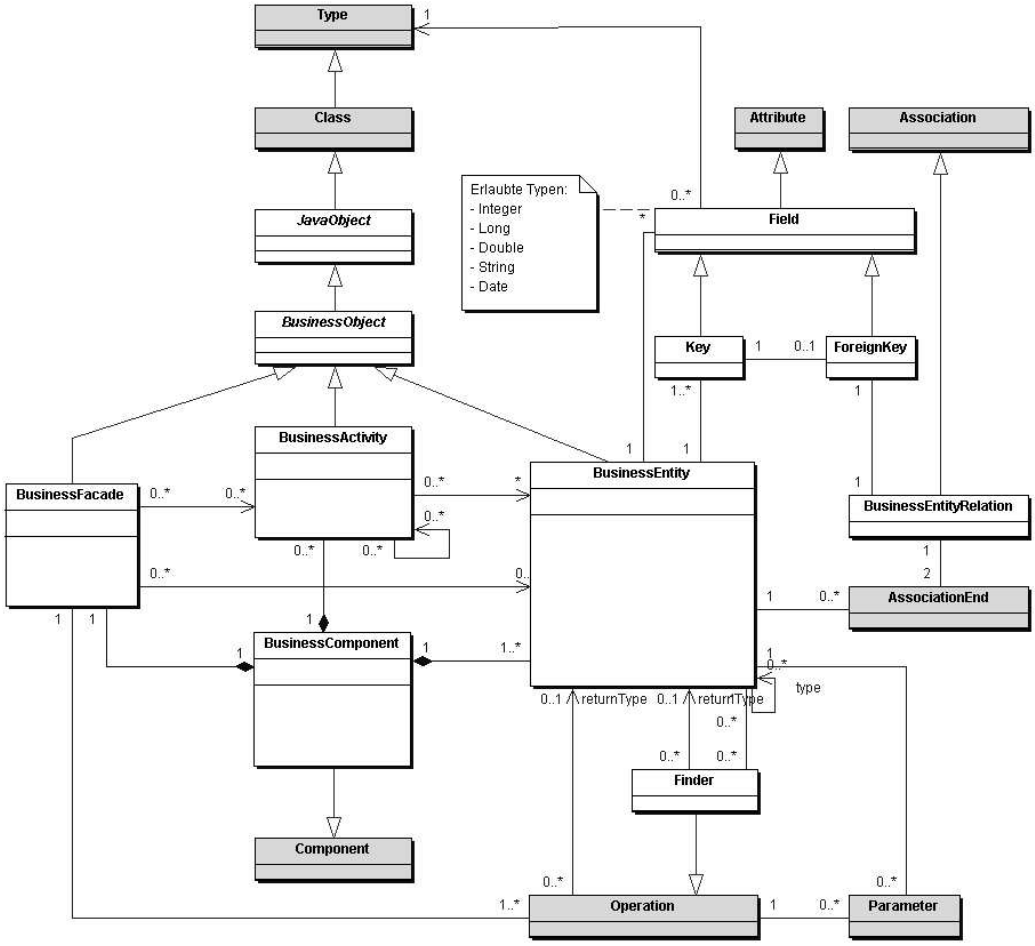

Abbildung 21: Das erweiterte Metamodell

7.6 Design

Das Design eines Systems oder einer Komponente teilt sich nach der MDA in ein fachliches und technisches Design auf. Beide befinden sich auf unterschiedlichen Abstraktionsebenen.

7.6.1 Fachliches Design

Das fachliche Design entspricht dem von der OMG spezifizierten PIM. Die XMI Repräsentation des PIM dient als Input für den MDA Generator. Als Rahmen des Designs der serverseitigen Architektur dient die BMW Component Architecture (CA, [Web02]). Diese gibt die Auszeichnung von Elementen des Designs sowie die Erweiterung des Metamodells vor. Für die Diplomarbeit wurden UML-Klassendiagramme verwendet, um das PIM der Architektur darzustellen. Die erforderlichen Modellierungselemente und deren Bezeichnungen ergaben sich aus dem Fachkonzept, das für ein Projekt oder System entworfen wurde. Das fachliche Design oder Modell sollte keinerlei technischen Informationen (z. B. für J2EE) beeinhalten und auf das Wesentliche reduziert sein.

Die im folgenden beschriebene Vorgehensweise hat keinen Anspruch auf Vollständigkeit und will auch nicht das komplexe Unterfangen eines Software-Entwicklungsprozesses von der Analyse über das Design bis zur Auslieferung beschreiben, sondern möglichst schnell zu einem verwendbaren PIM gelangen, das als Input für den b+m Generator dienen soll.
Aus dem Fachkonzept ergeben sich in der Regel die Subsysteme und Komponenten einer Anwendung, für die sich eine Generierung lohnt. Wenn man eine Komponente identifiziert hat, wird sie in einem UML Klassendiagramm als Component gezeichnet und mit dem Stereotyp <<BusinessComponent>> ausgezeichnet. Innerhalb der Komponente werden die weiteren Klassen modelliert. Hier unterscheidet man zwischen Klassen mittels unterschiedlicher Stereotypen <<BusinessEntity>>, <<BusinessActivity>> und <<BusinessEntity>>. BusinessEntity Klassen ergeben sich aus den Objekten eines Domänenmodells.
Die zwischen den Entitäten bestehenden Beziehungen werden als Assoziationen vom Stereotyp <<BusinessEnityRelation>> modelliert und mit einer entsprechenden Multiplizität und

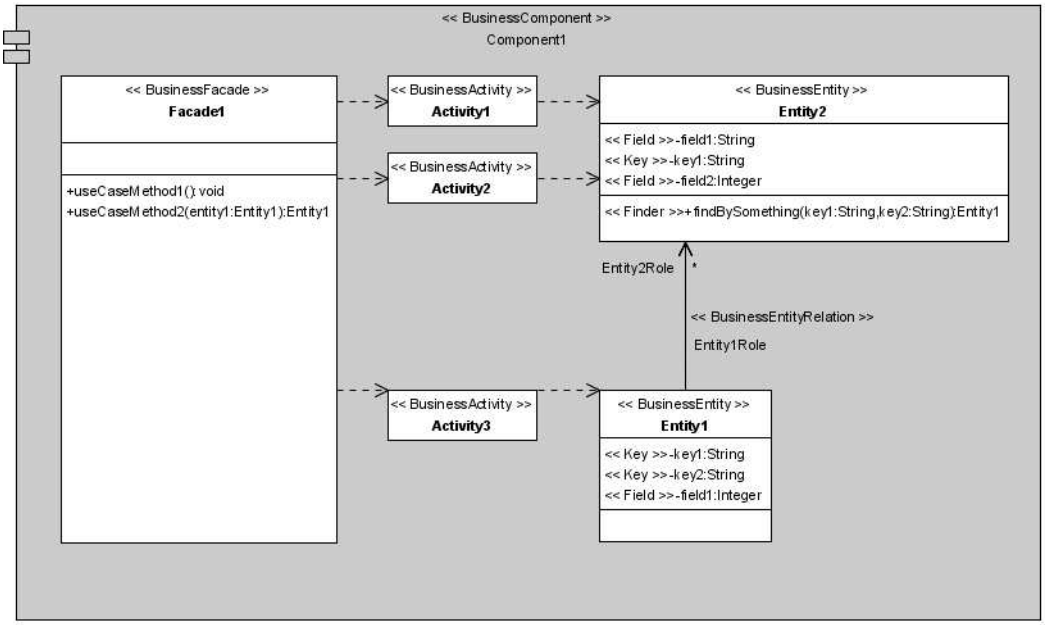

Abbildung 22: Ein beispielhaftes fachliches Design mit Poseidon UML

Rollennamen versehen.
Eine Klasse vom Stereotyp <<BusinessFacade>> definiert die Schnittstelle der Komponente. Sie enthält Methoden, die den Anwendungsfällen aus dem Fachkonzept entsprechen. Für die Ausführung der Anwendungsfälle fügt man Klassen vom Stereotyp <<BusinessActivity>> hinzu, die sich aus den Einzelaktivitäten eines Anwendungsfall ergeben. Es empfielt sich, zwischen BusinessFacade, BusinessActivities und BusinessEntities Abhängigkeiten in Form von Dependencies einzuzeichnen, um diese optisch zu dokumentieren und dem Generator weitere Anhaltspunkte für die Generierung zu geben. Ein solchermaßen erstelltes Modell genügt als Input für den MDA Generator und kann in das XMI Format exportiert werden.
Für die Diplomarbeit musste bei der Erstellung der BusinessEntities jedoch etwas anders vorgegangen werden, da für bereits existierende Datenbankmodelle generiert werden sollte, was eher einem Reengineering gleichkommt und für den MDA Ansatz sicherlich nicht optimal ist. Diese Vorgehensweise hat den Nachteil, dass manche technischen Details (z. B. rein

technische Helper Tabellen, Primärschlüssel/Fremdschlüssel Kombinationen), die beispielsweise der Optimierung der Datenbank dienen, sich nicht immer aus dem PIM ableiten lassen. Es ist also wesentlich einfacher, wenn man auf der "grünen Wiese" mit der Modellierung beginnt und daraus die Datenbanktabellen generiert. Ob diese dann auch performant sind, ist eine andere Frage.

7.6.2 Design Constraints

Über Design Constraints können Einschränkungen für den Entwurf eines PIM festgelegt werden. Ein UML Klassendiagramm erlaubt viele Möglichkeiten, wie eine Architektur modelliert werden kann, da es durch das allgemeine Metamodell der OMG beschrieben wird. Durch ein erweitertes Metamodell und die Anpassung an eine Problemdomäne schränkt man die Entwurfsmöglichkeiten auf ein sinnvolles Maß deutlich ein. So kann man beispielsweise verlangen, dass alle Klassen mit einem bestimmten Stereotypen ausgezeichnet werden müssen oder dass eine Klasse nur über eine Assoziation mit einer anderen Klasse verknüpft werden darf, wenn beide vom selben Stereotyp sind.
Durch Design Constraints verhindert man den Entwurf eines Modells, das nicht mit einer vorgegebenen Architekturfamilie konform ist und bei der Durchführung des Transformationsprozesses zu Fehlern führen würde. Die Einschränkungen ergeben sich direkt aus der Spezifikation der Architekturfamilie (hier durch die BMW Component Architecture ausgedrückt), für die generiert werden soll. Das Nichteinhalten der Vorgaben soll beim Transformationsprozess an den Designer über eine Fehler- oder Warnmeldung zurückgemeldet werden.
Das b+m Generator Framework unterstützt das programmatische Überprüfen von Design Constraints über die Erweiterung des Metamodells. Dazu muss die Methode `CheckConstraints()` der Klasse `de.bmiag.genfw.meta.Element` von einer Subklasse überschrieben werden. Da die Subklasse als Stereotyp im PIM verwendet werden kann, ist eine spezifische Zuordnung von Einschränkungen an beliebige Modellierungselemente möglich.
Die Klasse JavaObject überschreibt checkConstraints() und delegiert die Überprüfung an drei weitere abstrakte Methoden, die von den Subklassen (BusinessEntity, BusinessActivity, BusinessFacade) implementiert werden müssen:

- `checkAttributes()`
- `checkMethods()`
- `checkDependencies()`

In den Methoden findet dann die Überprüfung der jeweiligen Modellierungselemente statt.

```
protected void checkAttributes() throws DesignException {
      Iterator iter = Attribute().iterator();
      Attribute curAttr = null;
      boolean hasKey = false;
      while (iter.hasNext()) {
            curAttr = (Attribute) iter.next();
            if (!(curAttr instanceof Field)) {
                  throw new DesignException(
                        "Attribute "    + Name().toString()   + "."
                        + curAttr.Name().toString()
                        + " must be a <<Field>>.");
            }
```

```
            if (curAttr instanceof Key)
                  hasKey = true;
            }

            // at least one attribute must be a key in this entity
            if (!hasKey) {
                  throw new DesignException(
                        "At least one Attribute has to be a key in
                        + "BusinessEntity " + Name().toString() + ".");
            }
      }
}
```

Listing 5: Die Methode `checkAttributes()`

Es empfiehlt sich, Einschränkungen und Vorgaben für bestimmte Stereotypen zunächst textuell (z. B. in tabellarischer Form) festzulegen und diese anschließend in der Erweiterung des Metamodells auszuprogrammieren. Alternativ können die Einschränkungen auch direkt durch das Klassendiagramm des erweiterten Metamodells ausgedrückt werden und zwar insbesondere durch die Assoziationen zwischen den Klassen. So gibt die Multiplizität oder überhaupt erst die Verknüpfung mit einer anderen Klasse Klarheit darüber, was in welcher Weise erlaubt ist. Das Diagramm (vgl. Abbildung 21) kann dabei sehr genau und restriktiv gezeichnet sein, so dass kein Spielraum für Interpretationen bleibt.

So ergibt sich beispielsweise aus dem Diagramm, dass eine BusinessComponent mindestens eine BusinessEntity haben muss, oder dass Methoden einer BusinessEntity vom Stereotyp Finder sein müssen.

Aus dem Diagramm kann folgende textuelle Beschreibung für die zentralen Klassen abgeleitet werden:

BusinessComponent

Eine BusinessComponent ist eine Component und enthält genau eine BusinessFacade, mindestens eine BusinessEntity und beliebig viele BusinessActivities.

BusinessFacade

BusinessFacade ist ein BusinessObject und Bestandteil von BusinessComponent. Sie muss mindestens eine Operation haben. Der Rückgabetyp (returnType) der Operationen kann void oder vom Typ BusinessEntity (bzw. BusinessEntity[], wenn eine Liste zurückgegeben werden soll) sein. Es sind beliebig viele Parameter vom Typ BusinessEntity möglich. BusinessFacade kann beliebig viele Abhängigkeiten (Dependencies) zu BusinessActivities und BusinessEntities aufbauen.

BusinessActivity

BusinessActivity ist ein BusinessObject und Bestandteil von BusinessComponent. Sie kann beliebig viele Abhängigkeiten (Dependencies) zu anderen BusinessActivities oder zu BusinessEntities aufbauen.

BusinessEntity

BusinessEntity ist ein BusinessObject und Bestandteil von BusinessComponent. Sie hat mindestens einen Key und beliebig viele Fields. Typen für Fields können sein: Integer, Long, Double, String und Date. BusinessEntity kann beliebig viele AssociationEnds besitzen. Ein AssociationEnd muss sich auf genau eine BusinessEntityRelation beziehen. An eine BusinessEntityRelation ist genau ein ForeignKey gebunden. Dieser ist wiederum einem Key zugeordnet.
Operationen sind nur vom Subtyp Finder erlaubt. Ein Finder gibt eine oder einen Array mit Objekten vom Stereotyp BusinessEntity zurück. Es sind beliebig viele Parameter möglich, deren Typ die Namen Integer, Long, Double, String und Date speichern darf.

7.6.3 Technisches Design

Das technische Design ist in den Konstrukten und der Sprache der Zielplattform (J2EE) formuliert. Das Design beruht weitgehend auf der Anwendung von EJB Patterns [Mar02].

3-tier Client/Server Architektur

Client/Server Systemarchitekturen sind häufig mehrschichtig (n-tier Architekturen) aufgebaut, um sowohl eine logische wie auch technische Trennung von Aufgaben zu haben und eine große Anzahl von Abhängigkeiten zu vermeiden. Ganz grob teilt man solch eine Architektur zunächst in drei Schichten ein, die vor allem die physikalische Trennung (Client-Rechner, Application Server, Datenbank-Server) darstellen:

- **Presentation-Tier:** Die Frontend-Applikation auf dem Client-System, das für die Präsentation der Daten verantwortlich ist.
- **Business-Logic-Tier:** Enthält die fachlichen Komponenten, Services, Domänen Objekte. In einer J2EE-Architektur wird sie durch den Application Server repräsentiert.
- **Data-Tier:** Der permanente Datenspeicher, üblicherweise ein Datenbanksystem.

Die genannten Schichten können je nach Design noch weiter unterteilt werden, um die Architektur weiter zu verfeinern.

Die Business Logik Schicht lässt sich in vier technische Layers aufteilen:

- *Business Facade Schicht*, bestehend aus einer Stateless SessionBean Komponente, die fachlich zusammengehörende Anwendungsfälle in Form von fachlichen Methoden gruppiert und damit damit die Schnittstelle für den Klienten zur Verfügung stellt. Sie steuert den Workflow eines Anwendungsfalles und wird auch Use Case Controller genannt
- *Business Activity Schicht* bestehend aus einfachen Java Klassen, die wiederverwendbare Fachlogik ausführen (vgl. Command Pattern [Gam94]).
- *Data Access Object (DAO) Schicht* bestehend aus einfachen Java Klassen, die Aufrufe an die Entity Schicht delegiert. Im Gegensatz zur Vorgabe der CA wird hiermit eine zusätzliche, rein technische Schicht zwischen Facade bzw. BusinessActivities und EntityBeans gezogen, um unabhängig vom verwendeten Persistenzmechanismus zu sein. Man könnte hier beispielsweise die EntityBean Komponenten durch Java Data Objects (JDO) ersetzen, ohne die darüberliegenden Schichten ändern zu müssen.
- *Business Entity Schicht* bestehend aus EntityBean Komponenten

Für den Datenaustausch zwischen Präsentationsschicht und Business Logik Schicht werden Value Objects verwendet. In der Session Facade werden die Value Objects in Models umgewandelt. Models sind intelligente Datencontainer und unterstützen **C**reate, **R**ead, **U**pdate, **D**estroy (CRUD) Funktionalität. Sie werden von der Facade entweder direkt an ein Activity Object oder an eine Entity weitertransportiert.
In der Entity werden die Daten des Models ausgelesen und in der Datenbank persistiert. Als Persistenzmechanismus wird Container Managed Persistance (CMP) und Relations (CMR) von EntityBeans verwendet, der durch EJB 2.0 spezifiziert worden ist und auch durch die Weblogic Application Server Plattform ab der Version 8.1 verfügbar ist.

Eine klare Trennung und Aufteilung der Schichten ermöglicht eine bessere Wartung und Anpassbarkeit der Architektur. Es sind keine Aufrufe von niedrigen zu höheren Schichten erlaubt – das verhindert eine bunte Mischung und starke Kopplung der einzelnen Objekte untereinander. Ein Aufruf von oben nach unten darf Schichten überspringen mit Ausnahme der DAO Schicht. Diese Möglichkeit ergibt sich aus der CA Spezifikation [Web02], die Aufrufe aus der Facade direkt auf Business Entities (bzw. DAOs) erlaubt.

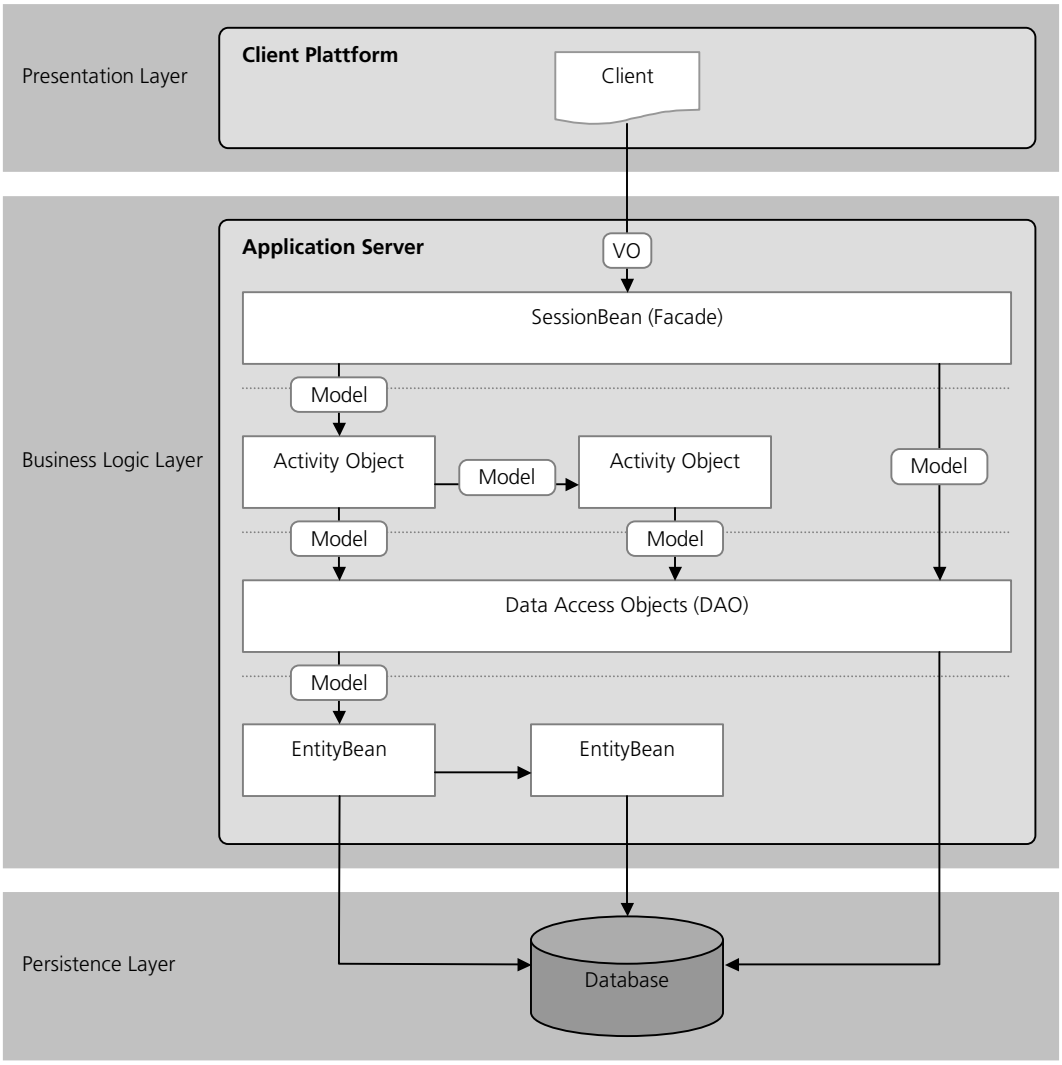

Abbildung 23: Das technische Design

Value Object

Das Value Object ist ein einfacher Datencontainer, der eine Ansammlung von Daten vom Klienten zum Server hin- und zurücktransportiert und damit Netzwerkzugriffe einspart. In der Business Facade werden die Daten des Value Objects über den UiMapper in ein Model Object kopiert. Damit ist eine klare Trennung zwischen dem Datenformat des Klienten und der des Servers möglich. Ein Value Object bildet wie das Model Object eine EntityBean Komponente ab. Diese Art von Value Object wird auch Domain Data Transfer Object genannt [Mar02]. Value Objects bilden Assoziationen zwischen EntityBeans im Gegensatz zu Model Objects und EntityBeans nicht über Objektreferenzen ab, sondern speichern vielmehr die ID des assoziierten Objekts in einem Fremdschlüssel-Attribut. Damit sind Value Objects näher am Datenbankmodell.

Model

Ein Model ist ein intelligenter Datencontainer, das die server-seitige Entsprechung eines Value Objects ist. Intelligent deshalb, weil es selbst CRUD Operationen zur Verfügung stellt. Intern ruft es dabei für die Ausführung Objekte der DAO Schicht auf. Models werden durch die Schichten transportiert, von der Business Facade über die Business Activity bis zur Business Entity. In der Business Entity Schicht werden sie mit den Business Entities synchronisiert, um die Aktualisierung oder Persistierung des Model auf der Datenbank zu ermöglichen.
Die Hierarchie bzw. das Beziehungsgeflecht von EntityBean Komponenten wird direkt auf die entsprechenden Models abgebildet.

Data Access Object

Ein Data Access Object dient der reinen Delegation von Aufrufen an die EntityBeans. Sie enthalten die Operationen `create()`, `load()`, `store()`, `remove()` sowie `findXXX()`, die ihrerseits die entsprechenden Operationen auf der EntityBean aufrufen.

Entity Bean

Eine EntityBean Komponente setzt sich zusammen aus

- EntityBean Klasse, implementiert die fachlichen Daten als Attribute
- Local/Remote Interface, Schnittstelle der Bean Komponente
- Local/Remote Home Interface, Schnittstelle der Fabrik zum Erzeugen von EJB Objects
- Business Interface für den Compile-Check (vgl. [Mar02])
- Primary Key Klasse, identifiziert jede EntityBean-Komponente

UiMapper

Der UiMapper ist eine Utility Klasse, die ein Value Object in ein Model (und umgekehrt) umwandelt. Im einfachsten Fall werden die Daten von einem Objekt in das andere Objekt kopiert. Im komplexeren Fall besteht die Aufgabe des UiMappers darin, Daten oder Datentypen des Klienten (Servers) an die des Servers (Klienten) anzupassen und zu formatieren, um die Anforderungen des jeweilig anderen zu erfüllen. Das Beziehungsgeflecht zwischen Models wird durch den UiMapper „flach“ gemacht, d. h. eine Assoziation zwischen zwei Models wird im Value Object durch Fremdschlüssel Attribute ausgedrückt.
Der UiMapper wird über ein UiService Objekt zur Verfügung gestellt.

Hier ein Auszug aus der Klasse DealerMasterdataUiMapper:

```
public IDomesticDealer mapDomesticDealerToUi(DomesticDealer model) {
      IDomesticDealer vo = new UiDomesticDealer();
      vo.setActiveDateTo(model.getActiveDateTo());
      vo.setDealerAssignment(model.getDealerAssignment());
      vo.setSulevFlag(model.getSulevFlag());
      vo.setQuotaDealerNumber(model.getQuotaDealerNumber());
      vo.setWholesalerID(model.getWholesalerID());
      vo.setWholesalerType(model.getWholesalerType());
      vo.setOrdererDomestic(model.getOrdererDomestic());
      return vo;
}
```

Listing 6: Die Methode `mapDomesticDealerToUi()`

Objekt Fabriken

Idealerweise besitzt jede der bereits erwähnten Schichten eine Fabrik, über die man Objekte der Schicht erzeugen kann.

Zum jetzigen Zeitpunkt können Objekte von vier Schichten erzeugt werden:

- Business Activity Factory (BaFactory), erzeugt Business Activity Objects
- Data Access Object Factory (DaoFactory), erzeugt Data Access Objects
- Home Factory (HomeFactory), erzeugt EntityBeans

Eine Ausnahme bildet das Erzeugen von Objekten der Business Facade Schicht. Diese werden über den Test Client erzeugt.

Die Erzeugung von Models und Value Objects ist bisher nicht über eine Fabrik vorgesehen, wird aber beispielsweise im DTO Factory Muster von [Mar02] vorgeschlagen.

Business Facade Session Bean

Die Business Facade ist eine SessionBean Komponente.

Sie setzt sich zusammen aus

- SessionBean Klasse, implementiert die fachlichen Operationen als Methoden
- Local/Remote Interface, Schnittstelle der Bean Komponente
- Local/Remote Home Interface, Schnittstelle der Fabrik zum Erzeugen von SessionBean Kompontenten
- Business Interface für den Compile-Check (vgl. [Mar02])
-

Die Business Facade implementiert den Workflow, der für die Ausführung eines Anwendungsfalls notwendig ist in fachlichen Operationen. Daten vom Klienten werden in Form eines Value Objects an die Methoden übergeben. Ein Value Object wird über einen UiMapper in ein Model gemapped. Die fachliche Methode übergibt das Model anschließend an eine oder mehrere Business Activity Objekte, die die Fachlogik ausführen. Hat eine Operation einen oder mehrere Rückgabewerte, so werden diese als Model an die Facade zurückgegeben und dort wieder auf entsprechende Value Objects gemapped.

Business Activity

Die Business Activity ist eine einfache Java Klasse, die dem Command Pattern [Gam94] folgt und wiederverwendbare Logik in einem Objekt kapselt. Sie hat getter und setter Methoden,

um Daten für die Ausführung der Logik bereitszustellen und das Ergebnis abfragen zu können. Die Daten werden in Form eines Model an die Activity übergeben. Der Code für die Ausführung selbst befindet sich in der `execute()` Methode. Dort können Objekte der Data Access Schicht aufgerufen werden, um auf die Datenbank zuzugreifen.

7.6.4 Mapping

Die mit Stereotypen markierten Modellierungselemente im PIM werden über die Templates auf plattformspezifische Artefakte gemapped. Aus den Klassen des PIM werden ein oder mehrere Artefakte generiert. Es gibt also ein 1:1 oder ein 1:N Mapping zwischen PIM und den generierten Artefakten.

Beispiel für 1:1: Eine Klasse Validate vom Stereotyp BusinessActivity des PIM wird auf genau eine Java Klasse ValidateBA gemapped.

Beispiel für 1:N: Eine Klasse DealerMasterData vom Stereotyp BusinessFacade des PIM wird auf eine Implementierungsklasse DealerMasterDataBFBean und drei Interfaces gemapped.

PIM Stereotyp	J2EE
BusinessComponent	Deployment Deskriptoren, EAR Archiv, HomeFactory, DaoFactory, BaFactory, UiMapper, UiService, BusinessFacadeTest
BusinessFacade	EJB Komponente (SessionBean, Remote/Local Interface, Business Interface, Remote/Local Home Interface)
BusinessActivity	Einfache Java Klasse
BusinessEntity	EJB Komponente (EntityBean, Remote/Local Interface, Business Interface, Remote/Local Home Interface), Primary Key Klasse, Model, Value Object, Value Object Interface, Data Access Object, Data Access Object Interface
Field	EntityBean Attribut (getter, setter), CMP Mapping im Deployment Deskriptor
Key	EntityBean Attribut (getter, setter), Primary Key Klasse Attribut
BusinessEntityRelation	EntityBean Attribut einer oder mehrerer assoziierten EntityBeans (getter, setter), CMR Mapping in Deployment Deskriptor
Finder	`findXXX()` Methode in EntityBean und Deployment Deskriptor

Tabelle 1: Zuordnung Stereotyp zu J2EE Konstrukt.

Dependencies werden aus dem Standard Metamodell verwendet und zu `import` Anweisungen von Klassen in der SessionBean und Business Activities transformiert.
Das Mapping der meisten Stereotypen ist trivial mit Ausnahme der BusinessEntityRelation. Für die Abbildung einer Assoziation auf eine Datenbankrelation muss etwas mehr Aufwand getrieben werden.

Abbildung von Assoziationen

Eine Beziehung zwischen zwei Klassen wird in UML durch eine Assoziation ausgedrückt und hat unter anderem folgende Eigenschaften

- Bezeichner
- Direktionalität (Navigierbarkeit und Sichtbarkeit der beiden teilnehmenden Klassen untereinander)
- Multiplizität (Anzahl der Objekte zur Laufzeit, die von der jeweilig anderen Klasse referenziert werden können)

Eine Assoziation ist mit einem Attribut einer Klasse verknüpft. Das Attribut speichert eine oder mehrere Referenzen von Objekten einer assoziierten Klasse. In der Objektorientierung spricht man dabei von sogenannter referentieller Identifikation (vgl. [Akc00])

Auf Datenbankebene wird eine Beziehung zwischen zwei Tabellen in der Regel durch eine Primärschlüssel – Fremdschlüssel Beziehung ausgedrückt. Diese Art der Referenzierung wird als assoziative Referenzierung [Akc00] bezeichnet. Jede Tabelle hat eine oder mehrere Spalten, die als Primärschlüsselspalte gekennzeichnet sind. Alle Werte dieser Primärschlüssel-Spalten zusammengenommen bilden die Eindeutigkeit eines Tabelleneintrags.
Eine Tabelle, die eine Beziehung zu einer anderen aufbauen will, muss eine oder mehrere Spalten haben, die als Fremdschlüssel-Spalte gekennzeichnet sind. In einer Fremdschlüsselspalte wird der Wert der Primärschlüsselspalte der referenzierten Tabelle gepeichert.
Im Gegensatz zum objektorientierten Programmparadigma spielen Namen und Direktionalität keine Rolle. Auf jede Tabelle, die eine Beziehung zu einer anderen hat, kann über eine entsprechend formulierte SQL Anweisung zugegriffen werden. Es gibt keine Kapselung.

Multiplizität/Kardinalität können auf Datenbankebene folgendermaßen ausgedrückt werden:

- ***1:1***: Ein Fremdschlüssel einer Tabelle A referenziert einen Primärschlüssel einer Tabelle B und umgekehrt. Der Fremdschlüssel jeder Tabelle kann dabei entweder gleichzeitig ihr Primärschlüssel sein oder durch eine eigene Tabellenspalte ausgedrückt werden. Im ersten Fall handelt es sich um eine Primärschlüssel/Fremdschlüssel-Kombination. Damit existiert für einen Eintrag einer Tabelle A genau ein Eintrag in einer Tabelle B. Diese Art der Beziehung ist in der Praxis eher seltener anzutreffen, da man beide Tabellen in einer einzigen zusammenfassen würde.
- ***1:N***: Ein Fremdschlüssel einer Tabelle A entspricht einem Primärschlüssel einer Tabelle B. Tabelle A ist dabei die N-Seite der Beziehung. Es gibt also mehrere Tabelleinträge in Tabelle A, die alle auf einen Eintrag der Tabelle B zeigen. Die 1:N Beziehung ist in der Praxis am häufigsten anzutreffen
- ***N:M***: Eine N:M Beziehung zwischen einer Tabelle A und Tabelle B wird durch eine Beziehungstabelle C ausgedrückt. C hat mindestens zwei Fremdschlüsselspalten, die gleichzeitig Primärschlüssel sind, in der die Primärschlüsselwerte von A und B gespeichert sind. C referenziert A und B über ihre Fremdschlüssel.

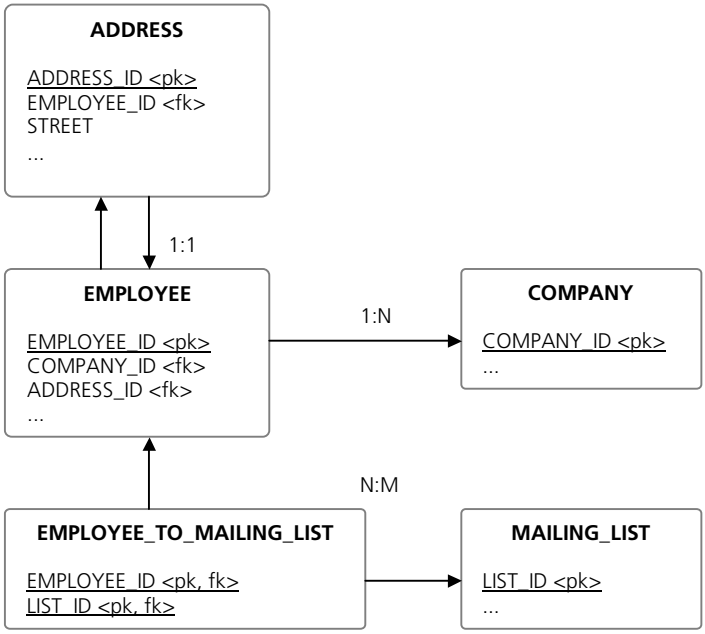

Abbildung 24: Beispiel mit 1:1, 1:N und N:M Kardinalitäten

Um nun eine Assoziation zwischen zwei Klassen im PIM auf die entsprechende Datenbankrelation abzubilden, muss das Metamodell ergänzt werden. Zunächst erweitert man die Klasse Assoziation mit der Subklasse BusinessEntityRelation und die Klasse Field mit der Subklasse ForeignKey. Ein ForeignKey Objekt referenziert ein Key Objekt. Einem BusinessEntityRelation Objekt können ein oder mehrere ForeignKey Objekte hinzugefügt werden, so wie es auf Datenbankebene auch mehrere Fremdschlüsselspalten in einer Tabelle geben kann. Damit hat man eine Primärschlüssel/Fremdschlüssel-Beziehung objektorientiert abgebildet und an ein Assoziationsobjekt gebunden.

Fremdschlüssel im PIM

Fremdschlüsselspalten in Datenbanktabellen werden **nicht** in das PIM direkt als Attribute übernommen, sondern als ForeignKey Objekt zur Generierungszeit an eine Assoziationsbeziehung gebunden. Ausnahme sind Spalten, die gleichzeitig als Primärschlüssel deklariert sind.

Auch wenn im PIM keine Fremdschlüsselattribute sichtbar sind, muss trotzdem ein logischer Attributname für Fremdschlüsselspalten vergeben werden, um die vollständige Generierung von Value Objects zu ermöglichen. Value Objects bilden im Gegensatz zu Models und EntityBeans Datenbanktabellen direkt ab, müssen also auch Attribute für die Fremdschlüssel implementieren. Diese Information kann in den Templates über das erweiterte Metamodell aus BusinessEntityRelation Objekten ausgelesen werden. Jede BusinessEntityRelation wird mit externen Mapping Informationen zu Primär-/Fremdschlüssel-Zuordnungen versorgt.

Das Problem ist nun, wie und an welcher Stelle man die technischen Primärschlüssel/Fremdschlüssel Beziehungen als Informationen in den Transformationsprozess miteinfließen lässt. Ein Fremdschlüssel ist ein rein technischer Aspekt und hat in einem Modell, das technologieunabhängig sein soll, nichts zu suchen.
Ein Lösungsvorschlag ist nun, eine Assoziation im PIM mit dem Stereotyp <<BusinessEntity-Relation>> zu kennzeichnen. Damit hat man in den Templates über das Metamodell Zugriff auf die Assoziation und die oben beschriebene Infrastruktur.
Was nun fehlt, sind Informationen über das Mapping zwischen Fremd- und Primärschlüssel, sowie die technischen Spaltennamen der Datenbanktabellen. Wie schon in Kapitel 7.4 beschrieben, können diese Information in der Datei map.xml bereitgestellt werden. Pro Relation gibt es eine oder mehrere Keymaps. Die Keymap ordnet einem Fremdschlüssel genau einen Primärschlüssel zu. Gleichzeitig werden die Attributnamen auf die technischen Namen der Datenbank abgebildet.

```
<relation>
      <relation-role-name>domesticDealer-groupDealer</relation-role-name>
      <key-map>
            <foreign-key>
                  <cmp-field>groupDealerID</cmp-field>
                  <dbms-column>NV_WS_ORDERER_ID</dbms-column>
            </foreign-key>
            <primary-key>
                  <cmp-field>groupDealerID</cmp-field>
                  <dbms-column>NV_WS_ORDERER_ID</dbms-column>
            </primary-key>
      </key-map>
      <key-map>
            <foreign-key>
                  <cmp-field>groupDealerType</cmp-field>
                  <dbms-column>NV_WS_ORDERER_CA</dbms-column>
            </foreign-key>
            <primary-key>
                  <cmp-field>groupDealerType</cmp-field>
                  <dbms-column>NV_WS_ORDERER_CA</dbms-column>
            </primary-key>
      </key-map>
</relation>
```

Listing 7: Das `<relation>` Element für map.xml

Der Zugriff auf die Relation aus dem Metamodell erfolgt über einen eindeutigen Namen, der sich aus der Kombination der Namen beider beteiligten Entitäten erzeugen lässt.
An dieser Stelle tritt jedoch ein weiteres Problem auf: Wie kann man eine Eindeutigkeit herstellen, sobald eine Entität mehrere Beziehungen zu einer anderen Entität besitzt, die alle unterschiedliche Rollen spielen? Die erste Überlegung ist, jede Assoziation im PIM mit einem eindeutigen Namen zu kennzeichnen. Das führt bei der Generierung der getter und setter Methoden für die referenzierten Objekte in den EJB EntityBean Komponenten zu unschönen Namensgebungen, insbesondere bei bidirektionalen Beziehungen, da eine EntityBean für die

jeweilig andere auch eine andere Rolle spielt. Es ist also notwendig, dass nicht für die Assoziationen Namen vergeben werden, sondern deren Enden namentlich gekennzeichnet werden. Dort kann man exakt festlegen, welche Rolle eine Entität für die jeweilig andere spielt.

Beispiel: Eine Band besteht aus mehreren Musikern. Jeder Musiker spielt ein anderes Instrument und hat damit eine andere Rolle. In EntityBeans ausgedrückt, hat die EntityBean Band mehrere bidirektionale Assoziationen zur EntityBean Musiker. Das Assoziationsende bei Musiker ist mit dem Rollennamen des Musikers („Schlagzeuger", „Saxophonist", „Gitarrist", usw.) gekennzeichnet, während die Band immer die Rolle "Band" für alle Musiker innehat.

Abbildung 25: Die Beziehungen zwischen Band und Musiker.

„band" und „saenger" bilden nun zusammen einen eindeutigen Namen der Assoziation und dienen während des Transformationsprozesses als Schlüssel für das Auffinden des entsprechenden technischen Primärschlüssel/Fremdschlüssel Mappings in map.xml.

```
<relation>
      <relation-role-name>band-saenger</relation-role-name>
      <key-map>
            <foreign-key>
                  <cmp-field>saengerID</cmp-field>
                  <dbms-column>SAENGER_ID</dbms-column>
            </foreign-key>
            <primary-key>
                  <cmp-field>musikerID</cmp-field>
                  <dbms-column>MUSIKER_ID</dbms-column>
            </primary-key>
      </key-map>
</relation>
<relation>
      <relation-role-name>band-gitarrist</relation-role-name>
      <key-map>
            <foreign-key>
                  <cmp-field>gitarristID</cmp-field>
                  <dbms-column>GITTARIST_ID</dbms-column>
            </foreign-key>
            <primary-key>
                  <cmp-field>musikerID</cmp-field>
                  <dbms-column>MUSIKER_ID</dbms-column>
```

```
            </primary-key>
        </key-map>
</relation>
...
```

Listing 8: Das <relation> Element in map.xml für Band und Musiker

7.7 Templates

Für die Generierung in den Quellcode für die Zielplattform (J2EE) wird der Template-Mechanismus (vgl. Kapitel 6.4) des b+m Generator Frameworks verwendet. Als Grundlage für die Template-Erstellung dient die Referenzimplementierung des Anwendungsdesigns.
Zunächst muss entschieden werden, welche Artefakte (Java Dateien, Konfigurationsdateien) der Referenzimplementierung für eine Generierung geeignet sind. Ausschlaggebend dafür sind zum einen immer wiederkehrende Muster in den Artefakten selbst und Muster der Architektur an sich.
Die Templates werden in den Unterkapiteln vorgestellt und haben einen stark referentiellen Charakter, was etwas langatmig wirkt, aber durchaus gerechtfertigt ist durch den großen Zeitaufwand, den die Erstellung der Templates einnahm.

Für die Generierung identifizierte Artefakte

Klassen und Interfaces

- BusinessFacade (EntityBean, Remote/Local Interface, Remote/Local Home Interface, Business Interface)
- BusinessActivity
- BusinessEntity (EntityBean, Remote/Local Interface, Remote/Local Home Interface, Business Interface, Primary Key Class)
- BusinessFacadeTest (JUnit Testcase)
- BaFactory
- HomeFactory
- DaoFactory
- Model
- DataAccessObject (DAO Klasse und DAO Interface)
- UiMapper
- UiService
- ValueObject (VO Klasse und VO Interface)
-

Deployment Deskriptoren

- Application Descriptor
- EJB Descriptor
- Weblogic Descriptors
-

Konfiguration

- map (O/R-Mapping)

Teile der aufgelisteten Artefakte lassen sich durch die Template-Sprache XPand soweit dynamisieren, dass die Architektur generisch wird.

Eine Template-Datei stellt den Namensraum zur Verfügung, in dem mehrere Templates definiert werden können. Es liegt also nahe, Klassen und Interfaces, die logisch zusammengehören (z. B. eine Komponente bilden) in einem Namensraum zu definieren. In diesem Fall macht es keinen Sinn, für jedes Template eine eigene Datei anzulegen.

Klassen und Interface Artefakte bestehen aus folgenden Bereichen, die sich für eine Dynamisierung eignen:

- *Package Bereich*: Package Struktur ist für das Projekt konfigurierbar.
- *import Bereich*: Wenn generierte Klassen andere generierte Klassen importieren und die Package Struktur konfigurierbar ist, ist auch der Import Pfad generierbar.
- *Attribute-Deklaration*: Attribut-Namen und -Typ sind variabel.
- *Typ-Definition* (Klassen, bzw. Interface-Name): Typ-Name ist variabel.
- *Methoden-Deklaration*: Methoden-Signaturen sind variabel.
- *Methoden-Implementierung*: Teile von Methodenimplementierungen lassen sich dynamisch generieren.

Im abstrakten Deployment Deskriptor lassen sich folgende Elemente variabel gestalten

- `<enterprise-beans>`: Name, Remote/Local Home Interface, PrimaryKey, Felder, Finder-Methoden (EJB-QL Anweisungen) von EntityBeans und SessionBeans
- `<relationships>`: Name, Rollen, Felder und Teilnehmer der Beziehung
- `<assembly-descriptor>`: Methodenzugriff
-

In den herstellerspezifischen Deployment Deskriptoren sind folgende Elemente interessant:

- `<weblogic-enterprise-bean>`: Name, JNDI Name des Home Interfaces
- `<weblogic-rdbms-bean>`: Name, Datenbankname, Tabellenname, Feld-Zuordnung
- `<weblogic-rdbms-relation>`: Name, Rollennamen, Tabellenspalten-Zuordnung (Primärschlüssel, Fremdschlüssel)

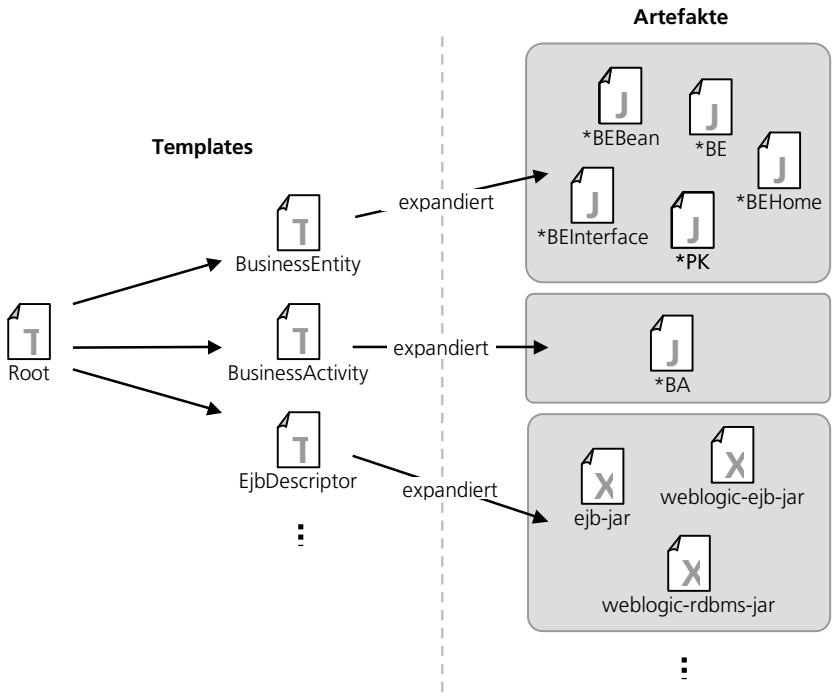

Abbildung 26: Kaskadierende Template-Expansion

Tabelle 2 zeigt die Zuordnung der Templates zu Namensräumen und die daraus generierten Artefakte

Namensraum	Templates	Artefakte
AppDescriptor.tpl	ApplicationXml	application.xml
BaFactory.tpl	BaFactoryClass	BaFactory.java
BusinessActivity.tpl	BusinessActivity	*BA.java
BusinessComponent.tpl	ExceptionClause ExceptionClauseList Local	-
BusinessFacade.tpl	SessionBeanClass SessionInterface BusinessInterface HomeInterface	*BFBean.java *BF.java I*BF.java *BFHome.java
BusinessFacadeTest.tpl	BusinessFacadeTestClass ProtectedRegion	*BFTest.java
BusinessEntity.tpl	EntityInterface BusinessInterface HomeInterface EntityBeanClass PrimaryKeyClass	*BE.java I*BE.java *BEHome.java *BEBean.java *PK.java
DaoFactory.tpl	DaoFactoryClass	DaoFactory.java

DataAccessObject.tpl	DataAccessObjectInterface DataAccessObjectClass	I*DAO.java *DAO.java
EjbDescriptor.tpl	MultiplicityMapper EjbJarXml WeblogicEjbJarXml WeblogicCmpRdbmsJarXml	 ejb-jar.xml weblogic-ejb-jar.xml weblogic-cmp-rdbms-jar.xml
HomeFactory.tpl	HomeFactoryClass	HomeFactory.java
JavaObject.tpl	InstanceVar PublicVar GetterSignature SetterSignature PublicGetterMethod PublicSetterMethod AllAttributesConstructor Visibility Signature (Operation) ParamSignature Signature (Parameter) ParamList (Operation) ParamList (Class) ToString ToStringWithKeys ProtectedRegion2	-
Map.tpl	MapXml	map.xml
Model.tpl	ModelClass	*.java
Root.tpl	Root (BusinessComponent) Root (BusinessFacade) Root (BusinessEntity) Root (BusinessActivity)	-
UiMapper.tpl	UiMapperClass	*UiMapper.java
UiService.tpl	UiServiceClass	*UiService.java
ValueObject.tpl	ValueObjectInterface ValueObjectClass	*Interface.java Ui*.java

Tabelle 2: Zurordnung der Namensräume zu den Templates

7.7.1 Root Templates

Im Namensraum Root sind die Root Templates definiert, die die Einstiegspunkte für den Generator zur hierarchischen Template Expansion bereitstellen. Root.tpl wird vor allen anderen Template-Dateien von dem Generator Framework geparst und ausgeführt.

Es sind insgesamt vier Einstiegspunkte vorgesehen:

- BusinessComponent: Expandiert die gesamte Komponente (Deployment Deskriptoren und die Templates der Klassen, die sich innerhalb der BusinessComponent befinden sowie Utilitiy und Helfer-Klassen)
- BusinessFacade: Expandiert die Templates der einzelnen Klassen und Interfaces, die zusammen die BusinessFacade Komponten bilden
- BusinessActivity: Expandiert die Templates der einzelnen Klassen und Interfaces, die zusammen die BusinessActivity Komponten bilden
- BusinessEntity: Expandiert die Templates der einzelnen Klassen und Interfaces, die zusammen die BusinessEntity Komponten bilden
-

Im Root Template für BusinessComponent wird auch das Einlesen der externen Datei (map.xml) für das Mapping der im PIM definierten Namen von Klassen, Attributen und Assoziationen angetriggert. Dies geschieht durch den Aufruf der Methode `BusinessComponent.map()`.

```
«DEFINE Root FOR BusinessComponent»
      «EXPAND Map::MapXml»

      «REM -- maps all database relevant names to the business entities: --»
      «REM --  DB column names to entity attributes --»
      «REM --  DB table name to entity name --»
      «map»

      «REM -- Deployment Descriptors --»
      «EXPAND AppDescriptor::ApplicationXml»
      «EXPAND EjbDescriptor::EjbJarXml»
      «EXPAND EjbDescriptor::WeblogicCmpRdbmsJarXml»
      «EXPAND EjbDescriptor::WeblogicEjbJarXml»

      «EXPAND UiMapper::UiMapperClass»
      «EXPAND UiService::UiServiceClass»

      «REM -- expand all factory templates for the component --»
      «EXPAND DaoFactory::DaoFactoryClass»
      «EXPAND BaFactory::BaFactoryClass»
      «EXPAND HomeFactory::HomeFactoryClass»
      «EXPAND Root FOREACH BusinessEntity»
      «EXPAND Root FOREACH CompositeBusinessEntity»
      «EXPAND Root FOREACH BusinessActivity»
      «EXPAND Root FOREACH BusinessFacade»
      «EXPAND BusinessFacadeTest::BusinessFacadeTestClass»
«ENDDEFINE»

«DEFINE Root FOR BusinessFacade»
      «EXPAND BusinessFacade::SessionBeanClass»
      «EXPAND BusinessFacade::HomeInterface»
      «EXPAND BusinessFacade::BusinessInterface»
```

```
      «EXPAND BusinessFacade::SessionInterface»
«ENDDEFINE»

«DEFINE Root FOR BusinessActivity»
      «EXPAND BusinessActivity::BusinessActivity»
«ENDDEFINE»

«DEFINE Root FOR BusinessEntity»
      «EXPAND BusinessEntity::BusinessInterface»
      «EXPAND BusinessEntity::EntityInterface»
      «EXPAND BusinessEntity::HomeInterface»
      «EXPAND BusinessEntity::EntityBeanClass»
      «EXPAND BusinessEntity::PrimaryKeyClass»
      «EXPAND ValueObject::ValueObjectInterface»
      «EXPAND ValueObject::ValueObjectClass»
      «EXPAND Model::ModelClass»
      «EXPAND DataAccessObject::DataAccessObjectInterface»
      «EXPAND DataAccessObject::DataAccessObjectClass»
«ENDDEFINE»

«DEFINE Root FOR CompositeBusinessEntity»
      «EXPAND CompositeModel::CompositeModelClass»
      «EXPAND CompositeValueObject::CompositeValueObjectClass»
      «EXPAND CompositeValueObject::CompositeValueObjectInterface»
«ENDDEFINE»
```

Listing 9: Root.tpl definiert die Root-Templates

7.7.2 BusinessFacade Templates

Der Namensbereich BusinessFacade ist der Klasse BusinessFacade im erweiterten Metamodell zugeordnet und besteht aus vier Templates, die zusammen die EJB SessionBean Komponente bilden.

SessionBeanClass

Das SessionBeanClass Template expandiert zu *BFBean.java. Die SessionBean implementiert die fachlichen Methoden, die zur Ausführung der Anwendungsfälle notwendig sind. Das Template ermittelt und generiert alle Methoden mit ihren Rückgabe- und Parametertypen, die in einer Klasse vom Stereoyp <<BusinessFacade>> im PIM definiert worden sind.
Da die Methoden die Steuerung des Workflows eines Anwendungsfalles implementieren sollen, der nicht generiert werden kann, sind geschützte Bereiche vorgesehen, die manuelles Hinzufügen von Code ermöglichen.

BusinessInterface

Das SessionBeanClass Template expandiert zu *BFInterface.java. Die Deklaration der Methoden ergibt sich in gleicher Weise wie für das SessionBeanClass Template beschrieben.

Das Business Interface kann für den „Local“ oder den „Remote“ Einsatz generiert werden. Der Unterschied drückt sich in den Methodensignaturen aus: Methoden für „Remote“ Komponenten müssen nach der EJB Spezifikation eine `java.rmi.RemoteException` werfen.

SessionInterface

Das SessionInterface Template expandiert zu *BF.java. Wie im BusinessInterface Template muss zwischen „Local“ und „Remote“ Generierung unterschieden werden, was sich in der Vererbungsbeziehung des Interfaces äussert (... `extends EJBLocalObject oder EJBObject`).

HomeInterface

Das HomeInterface Template expandiert zu *BFHome.java. Hier ist ebenfalls entscheidend, ob lokal oder remote generiert werden soll, was Auswirkung auf die Vererbungsbeziehung und Methodensignaturen hat.

Beispiel: Im PIM ist die Klasse DealerMasterdata mit dem Stereotyp <<BusinessFacade>> ausgezeichnet. Bei der Anwendung der beschriebenen Templates während des Transformationsprozesss werden aus dieser Klasse die Artefakte

- DealerMasterdataBFBean.java
- DealerMasterdataBFInterface.java
- DealerMasterdataBF.java
- DealerMasterdataBFHome.java

generiert.

7.7.3 BusinessEntity Templates

Der Namensbereich BusinessEntity ist der Klasse BusinessEntity im erweiterten Metamodell zugeordnet und besteht aus fünf Templates, die zusammen die EJB EntityBean Komponente bilden. Die generierten Artefakte sind vollständig, es sind keine geschützten Code-Bereiche vorgesehen.

EntityBeanClass

Das EntityBeanClass Template expandiert zu *BEBean.java. Es werden in variabler Weise alle getter und setter Methoden für Attribute hinzugefügt, die im PIM für die BusinessEntity angegeben sind. Assoziationen zu anderen BusinessEntities müssen ebenfalls über getter und setter Methoden ausgedrückt werden. Dies ist dann der Fall, wenn sie navigierbar sind. Die Multiplizität wird geprüft über das Assoziationsende, auf der sich die BusinessEntity befindet. Abhängig davon ergibt sich der Rückgabe-Typ bzw. Parameter-Typ und der Name der getter und setter Methoden. Ist die BusinessEntity bei 1:N auf der N Seite, wird der Name „Collection“ an den Methodennamen angehängt und eine `java.util.Collection` zurückgegeben bzw. als Parameter übergeben.
Eine Entitybean kann über die Methoden `getModel()` und `setModel()` ein Model speichern und zurückgeben. Das Model muss mit den Attributen der EntityBean synchronisiert werden. Da die Attribute variabel sind, muss auch die Synchronisierung zwischen Model und EntityBean variabel durch die Template Sprache XPand ausgedrückt werden. Zusätzlich muss das Beziehungsgeflecht zwischen BusinessEntities auch auf Model-Ebene synchronisiert werden.

BusinessInterface

Das BusinessInterface Template expandiert zu *BEInterface.java. Das Hinzufügen der Methoden der fachlichen Attribute, Assoziationen zu anderen BusinessEntities und das Setzen des Models ergibt sich in gleicher Weise wie im SessionBeanClass Template. Das Business Interface kann für den „Local" oder den „Remote" Einsatz generiert werden. Der Unterschied drückt sich in den Methodensignaturen aus: Methoden für Remote Komponenten müssen eine `java.rmi.RemoteException` werfen.

EntityInterface

Das EntityInterface Template expandiert zu *BE.java. Wie im BusinessInterface Template muss zwischen „Local" und „Remote" Generierung unterschieden werden, was sich in der Vererbungsbeziehung des Interfaces äussert (... `extends EJBLocalObject oder EJBObject`).

HomeInterface

Das HomeInterface Template expandiert zu *BEHome.java. Hier ist ebenfalls entscheidend, ob „Local" oder „Remote" generiert werden soll, was Auswirkung auf die Vererbungsbeziehung und Methodensignaturen hat.
Es werden alle `findXXX()` Methoden generiert, die für die BusinessEnitity im PIM angeben worden sind.

PrimaryKeyClass

Das PrimaryKeyClass Template expandiert zu *PK.java. Die Primary Key Klasse einer EJB EntityBean Komponente muss die Methoden `equals()` und `hashCode()` überschreiben. Die Eindeutigkeit ergibt sich aus einem Attribut oder einer Kombination von Attributen, die in der BusinessEntity mit dem Stereotyp <<Key>> ausgezeichnet worden sind.

Beispiel

Im PIM ist die Klasse GroupDealer mit dem Stereotyp <<BusinessEntity>> ausgezeichnet. Bei der Anwendung der beschriebenen Templates während des Transformationsprozesss werden aus dieser Klasse die Artefakte

- GroupDealerBEBean.java
- GroupDealerBEInterface.java
- GroupDealerBE.java
- GroupDealerBEHome.java
- GroupDealerPK.java

generiert.

7.7.4 BusinessActivity Template

Der Namensbereich BusinessActivity ist der Klasse BusinessActivity im erweiterten Metamodell zugeordnet und besteht aus einem einzigen Template, das zu *BA.java expandiert. Eine BusinessActivity kann Abhängigkeiten (Dependencies) zu anderen BusinessActivities oder BusinessEntities haben, die im Template als `import` Anweisung der Klasse, von der die BusinessActivity abhängt, ausgedrückt werden. BusinessActivities kapseln Fachlogik, die nicht generiert wird, sondern von Entwicklern manuell hinzugefügt werden muss. Aus diesem Grund sind in der generierten Klasse an verschiedenen Stellen geschützte Bereiche vorgesehen, in die Code eingefügt werden kann.

Beispiel

Im PIM wurden die Klassen AddGroupDealer und ValidateGroupDealer mit dem Stereotyp <<BusinessActivity>> ausgezeichnet. Bei der Anwendung des beschriebenen Templates während des Transformationsprozesss werden aus den beiden Klassen die Artefakte

- AddGroupDealerBA.java
- ValidateGroupDealerBA.java

generiert.

7.7.5 Model

Der Namensbereich Model besteht aus einem einzigen Template ModelClass und hat keine direkte Entsprechung im erweiterten Metamodell, sondern wird im Zusammenhang mit den BusinessEntity Templates expandiert. Das Template expandiert zu *.java, wobei * der Name der BusinessEntity im PIM ist.

Das generierte Artefakt ist vollständig, es sind keine geschützten Code-Bereiche vorgesehen. Das Model ist ein Datencontainer, der die selben fachlichen Attribute und Assoziationen wie die EntityBean Komponente besitzt. Deswegen entspricht die Template Implementierung weitgehend der des EntityBeanClass Templates.

Ein Model besitzt Lifecycle Methoden (`create()`, `store()`, `load()`, `remove()`), die über die DaoFactory auf Data Access Objects zugreifen. In `load()` müssen auch Assoziationen des Models zu anderen Models in variabler Weise berücksichtigt werden, um das Beziehungsgeflecht von BusinessEntities abzubilden.

Beispiel

Im PIM ist die Klasse GroupDealer und DomesticDealer mit dem Stereotyp <<BusinessEntity>> ausgezeichnet. Bei der Anwendung des beschriebenen Templates während des Transformationsprozesss werden die Artefakte

- GroupDealer.java
- DomesticDealer.java

generiert.

7.7.6 ValueObject

Der Namensbereich ValueObject besteht aus zwei Templates und hat keine direkte Entsprechung im erweiterten Metamodell, sondern wird im Zusammenhang mit den BusinessEntity Templates expandiert.

Das generierte Artefakt ist vollständig, es sind keine geschützten Code-Bereiche vorgesehen.

ValueObjectInterface

Das ValueObjectInterface Template expandiert zu I*.java, wobei * der Name der BusinessEntity ist.

Die getter und setter Methoden ergeben sich aus den Attributen, die für die BusinessEntity im PIM definiert sind. Assoziationen zwischen BusinessEntities werden in den Value Objects nicht weitergeführt, sondern in Attribute augelöst. Diese ergeben sich aus dem Fremdschlüsselattributen der assoziierten BusinessEntities.

ValueObjectClass

Das ValueObjectInterface Template expandiert zu Ui*.java, wobei * der Name der BusinessEntity ist.

Die variablen Bereiche werden wie im ValueObjectInterface Template in XPand implementiert.

Beispiel

Im PIM ist die Klasse GroupDealer mit dem Stereotyp <<BusinessEntity>> ausgezeichnet. Bei der Anwendung der beschriebenen Templates während des Transformationsprozesss werden aus dieser Klasse die Artefakte

- IGroupDealer.java
- UiGroupDealer.java

generiert.

7.7.7 DataAccessObject

Der Namensbereich DataAccessObject besteht aus zwei Templates und hat keine direkte Entsprechung im erweiterten Metamodell, sondern wird im Zusammenhang mit den BusinessEntity Templates expandiert.
Das generierte Artefakt ist vollständig, es sind keine geschützten Code-Bereiche vorgesehen.

DataAccessObjectInterface

Das DataAccessObjectInterface Template expandiert zu I*DAO.java, wobei * der Name der BusinessEntity ist. Lifecycle Methoden (`create()`, `load()`, usw) und `findXXX()` werden aus dem Namen und den mit dem Stererotyp <<Finder>> ausgezeichneten Methoden der zugeordneten BusinessEntity erzeugt.

DataAccessObjectClass

Das ValueObjectClass Template expandiert zu *DAO.java, wobei * der Name der BusinessEntity ist.
Die variablen Bereiche werden wie im DataAccesObjectInterface Template in XPand implementiert.

Beispiel

Im PIM ist die Klasse GroupDealer mit dem Stereotyp <<BusinessEntity>> ausgezeichnet. Bei der Anwendung der beschriebenen Templates während des Transformationsprozesss werden aus dieser Klasse die Artefakte

- IGroupDealerDAO.java
- GroupDealerDAO.java

generiert.

7.7.8 BusinessFacadeTest

Der Namensbereich BusinessFacadeTest besteht aus einem Template BusinessFacadeTestClass und hat keine direkte Entsprechung im erweiterten Metamodell, sondern ist an BusinessComponent gebunden.
Das Template expandiert zu *BFTest.java, wobei * der Name der BusinessFacade ist. Es generiert Testmethoden für jede vorhandene Methode, die einer Klasse vom Stereotyp BusinessFacade. Die Testmethoden sollen die Fachlogik für die Ausführung eines Tests überprüfen, die nicht generiert wird. Deshalb enthält das Template geschützte Bereiche, um das manuelle Hinzufügen von Code zu ermöglichen.

Beispiel:

Im PIM ist die Klasse DealerMasterdata mit dem Stereotyp <<BusinessFacade>> ausgezeichnet. Bei der Anwendung der beschriebenen Templates während des Transformationsprozesss wird aus dieser Klasse das Artefakt

- DealerMasterdataBFTest.java

generiert.

7.7.9 UiMapper und UiService

UiMapper und UiService sind zwei getrennte Namensräume (UiMapper.tpl, UiService.tpl). Für jeden Namensraum ist ein Template definiert. Beide Templates sind an BusinessComponent gebunden.

UiMapperClass

Das UiMapper Template expandiert zu *UiMapper.java, wobei * der Name der BusinessComponten ist. Der UiMapper wird in der Session Facade verwendet und konvertiert Value Objects zu Models und umgekehrt. Das Template erzeugt für jede Klasse vom Stereotyp BusinessEntity eine `mapXXX()` Methode, die die entsprechenden Value Objects und Models konvertiert.

UiServiceClass

Das UiMapper Template expandiert zu *UiService.java, wobei * der Name der BusinessComponten ist. Der UiService stellt den UiMapper für die Session Facade zur Verfügung. Die UiService Klasse wurde als Template implementiert, um es namentlich an die Komponente anzupassen.

Beispiel:

- DealerMasterdataUiMapper.java
- DealerMasterdataUiService.java

7.7.10 Deployment Deskriptor Templates

Für das Deployment im Weblogic Application Server werden vier Deployment Deskriptoren expandiert, für die in den Namenräumen AppDescriptor und EjbDescriptor je ein Template definiert ist. Jedes Template ist an BusinessComponent gebunden.

AppDescriptor

Der AppDescriptor Namensraum enthält das AppDescriptor Template, das zu application.xml expandiert. Der Application Deployment Deskriptor definiert ein Modul, das der BusinessComponent entspricht. Im Application Descriptor ist lediglich der Name des Moduls (*.ejb.jar) dynamisch, das die Implementierung der EJB Komponenten enthält.

EjbDescriptor

Der EjbDescriptor Namensraum enthält drei Templates

- EjbJarXml: Expandiert zu ejb-jar.xml
- WeblogicEjbJarXml: Expandiert zu weblogic-ejb-jar.xml
- WeblogicCmpRdbmsJarXml: Expandiert zu weblogic-cmp-rdbms-jar.xml

7.7.11 Utility Templates

Einige Templates haben Utility-Funktion und können von anderen Templates verwendet werden. Zum einen kann so in gewissen Bereichen die Konsistenz bei der Expansion gewahrt und zum anderen Änderungen an zentraler Stelle vorgenommen werden. Es gibt zwei Namensräume JavaObject und BusinessComponent, in denen mehrere dieser Templates definiert sind.

JavaObject

Der Namensraum JavaObject definiert die folgenden Templates

- diverse Helper für Attribute und Operationen: Diese Templates erzeugen verschiedene Methodensignaturen für Attribute und ergänzen Attributnamen
- ToString: Erzeugt eine `toString()` Methode, die alle Attribute einer Klasse über ein `System.out.println()` ausgibt
- ToStringWithKeys: Erzeugt eine `toString()` Methode, die alle als Schlüssel gekennzeichnete Attribute einer Klasse über System.out.println() ausgibt
- ProtectedRegion2: Erzeugt den geschützten Bereich mit Kommentaren und zwei IDs, die als Parameter an das Template übergeben werden können

BusinessComponent

Der Namensraum BusinessComponent definiert drei Templates, die im Zusammenhang mit der wahlweisen Generierung von „Local“ oder „Remote“ EJB Komponenten zusammenhängen

- ExceptionClause: Expandiert zu "throws RemoteException" bei „Remote“ Generierung
- ExceptionClauseList: Expandiert zu ", throws RemoteException" bei „Remote“ Generierung
- Local: Ein Flag, das die „Local“ oder „Remote“ Generierung konfiguriert

Map

Der Namensraum Map definiert ein Template MapXml, das zu map.xml expandiert. Das Template erstellt den PIM Anteil (Namen der BusinessEnities, deren Attribute und Beziehungen) der O/R Mapping Information und stellt geschützte Bereiche zur Verfügung, die mit den technischen Informationen des Datenbankmodells ergänzt werden können.

7.7.12 Factory Templates

Für die Factory Klassen HomeFactory, DaoFactory und BaFactory sind ein jeweils eigener Namensraum (HomeFactory.tpl, DaoFactory.tpl, BaFactory.tpl) mit je einem Template vorgesehen. Die Templates sind an BusinessComponent gebunden.

HomeFactoryClass

Das HomeFactoryClass Template expandiert zu HomeFactory.java. Das Template wird benötigt, um abhängig von den Konfigurationseinstellungen Local oder Remote Interfaces für die EntityBeans zu generieren. Dazu wird die Signatur der Methode `getHome()` variabel generiert.

DaoFactoryClass

Das DaoFactoryClass Template expandiert zu DaoFactory.java und erzeugt für jede BusinessEntity der BusinessComponent eine Fabrikmethode, die ein Data Access Objects zurückgibt.

BaFactoryClass

Das BaFactoryClass Template expandiert zu BaFactory.java und erzeugt analog zum DaoFactoryClass Template für jede Klasse vom Stereotyp BusinessActivity eine Fabrikmethode, die ein BusinessActivity Objekt zurückgibt.

7.8 Unique Identifiers (UIDs)

7.8.1 Problemstellung

Die aus dem PIM generierten Artefakte enthalten geschützte Bereiche (vgl. Kapitel 6.5), die bei wiederholter Generierung nicht überschrieben werden dürfen. Die geschützten Bereiche müssen mit einer eindeutigen Identifikation (Unique ID oder UID) gekennzeichnet sein, ansonsten kann der Generator die Bereiche nicht voneinander unterscheiden. Ein weitere Anforderung an die UID ist nicht nur die Eindeutigkeit, sondern auch Konstantheit über mehrere Generierungsprozesse hinweg. Sobald sich eine UID ändert, ist der geschützte Bereich nicht mehr zuordbar und wird überschrieben.

Wie erzeugt man diese UIDs?

Die Erzeugung der Artefakte ist direkt an die Modellierungselemente des PIM gebunden. Es liegt also nahe, die Modellierungselemente selbst mit einer UID auszuzeichnen, die sich dann bei der Transformation über das Metamodell in den Templates erfragen lässt. Nun kann man sich leicht vorstellen, dass es recht umständlich wäre, von Hand jedes Modellierungselement nach dem Zeichnen mit einer UID zu versehen. Hierbei kann man jedoch mit Unterstützung des UML Werkzeugs rechnen. Poseidon UML generiert für jedes neu gezeichnete Modellierungselement eine UID. Beim Export in XMI wird die UID im xmi.id Attribut abgelegt.

Nach mehreren Generierungsprozessen und Änderungen des PIMs während der Diplomarbeit hat sich jedoch herausgestellt, dass Poseidon UIDs nicht konstant hält und bei Änderungen im Diagramm neu anlegt. Dies hat zum Verlust der geschützten Bereiche und dem darin manuell hinzugefügten Programmcode geführt.

Es musste deshalb nach einer anderen Lösung für die Erzeugung der UIDs gesucht werden

7.8.2 IDGenerator

Der IDGenerator ist ein kleines für die Diplomarbeit geschriebenes Java-Programm, das Dateien im XMI Format einliest, UIDs erzeugt und sie Elementen innerhalb der XMI Datei als Markierung (Tagged Value) zuordnet. Es ist Bestandteil des Eclipse-Projekts "MDA-Generator" und befindet sich dort in dem Unterverzeichnis /util im Package `de.softlab.mda.util.idgenerator`.

Das grundlegende Prinzip des XMI Formats

Alle durch das Metamodell beschriebenene Modellierungselemente für bestimmte Diagramme werden als XML Tags ausgedrückt. Bestimmte Eigenschaften der Elemente sind durch Attribute dargestellt. Der Zusammenhang zwischen Modellierungselementen (Assoziationen im Metamodell) ergibt sich durch die hierarchische Anordnung der XML Elemente untereinander.

XMI Export Format von Poseidon UML

Das XMI Format von Poseidon definiert folgende grundlegende Struktur. Laut Gentleware AG richtet sich das Format nach der aktuellen XMI Spezifikation der OMG.

```
<XMI>
      <XMI.header>
      </XMI.header>
      <XMI.content>
            <UML:Model>
                  <UML:Namespace.ownedElement>
                        <UML:Class/>
                        <UML:Package/>
                        <UML:Component/>
                        <UML:Stereotype/>
                        <UML:Dependency/>
                        <UML:Association/>
                        <UML:TagDefinition/>
                  </UML:Namespace.ownedElement>
            </UML:Model>
      </XMI.content>
</XMI>
```

XMI setzt einen ausgeprägten Referenzierungsmechanismus zur Vermeidung von Redundanzen ein. Redundanzen entstehen in UML Diagrammen beispielsweise, wenn mehrere Attribute einer Klasse vom selben Typ sind. Eine simple Serialisierung in XML würde dazu führen, dass der Typ jedes Attributes jedesmal neu definiert werden würde. Um das zu verhindern, wird der Typ in der XML Datei ein einziges Mal definiert und dann von anderen Elementen referenziert. Es existiert also eine Art zentrales Repository für Modellierungselemente, die mehrmals verwendet werden.

Dies führt insgesamt zu wesentlich kleineren Dateien, erschwert aber die Lesbarkeit für den Menschen. Die Struktur von XMI kann sehr komplex sein, da das Schachteln von Elementen auf die Spitze getrieben werden kann. Das XMI Format ist deswegen vor allem für die maschinelle Weiterverabeitung (z. B. durch Werkzeuge und Parser) geeignet und nicht für ma-

nuelle Änderungen. Ein weiteres Problem ist die Inkompatibilität der XMI Schnittstelle der UML Case Werkzeuge. Hier scheint jeder Hersteller sein eigenes Süppchen zu kochen, was die Austauschbarkeit von XMI Dateien unter den Anwendungen sehr erschwert und auch MDA Generatoren zu einem flexiblen Import Mechanismus zwingt.

```
<UML:Class xmi.id = 'lsm:ae1cf:f988cb67b2:-7f85'
           name = 'GroupDealer'
           visibility = 'public'
           isSpecification = 'false'
           isRoot = 'false'
           isLeaf = 'false'
           isAbstract = 'false'
           isActive = 'false'>
     <UML:Classifier.feature>
           <UML:Attribute xmi.id = 'lsm:ae1cf:f988cb67b2:-7f96'
                      name = 'groupDealerID'
                      visibility = 'private'
                      isSpecification = 'false'
                      ownerScope = 'instance'>
                <UML:StructuralFeature.type>
                      <UML:Class xmi.idref = 'lsm:ae1cf:f988cb67b2:-7f12'/>
                </UML:StructuralFeature.type>
           </UML:Attribute>
     </UML:Classifier.feature>
</UML:Class>
```

Listing 10: XMI Darstellung einer Klasse.

Oben abgebildet ist die XMI Darstellung der Klasse GroupDealer mit einem Attribute groupDealerID vom Typ String. Der Typ des Attributs wird über das xmi.idref Attribut referenziert. Wenn man die Referenz weiter verfolgt, entdeckt man die Definition der Klasse String:

```
<UML:Class xmi.id = 'lsm:ae1cf:f988cb67b2:-7f12'
           name = 'String'
           isSpecification = 'false'
           isRoot = 'false'
           isLeaf = 'false'
           isAbstract = 'false'
           isActive = 'false'>
</UML:Class>
```

Listing 11: Definiton der Klasse String in XMI

Idee

Wenn man sich auf den Werkzeug-Hersteller bei der Vergabe von UIDs nicht verlassen kann und es nicht möglich ist, programmatisch direkt in das Werkzeug einzugreifen, muss man am XMI Export des Diagramms ansetzen. Die Idee ist nun, den XMI Export des UML Werkzeugs zu parsen und diejenigen Elemente mit einer UID zu ergänzen, die für das Identifizie-

ren von geschützten Bereichen wichtig sind. Es muss jedoch gewährleistet sein, dass beim Import der XMI Datei in das UML Werkzeug extern ergänzte UIDs nicht gelöscht oder überschrieben werden, d. h. Attribute wie beispielsweise xmi.id können nicht verwendet werden. An dieser Stelle kommt der Erweiterungsmechanismus von UML Diagrammen ins Spiel, wie er in Kapitel 3.6 bereits beschrieben wird. Modellierungselemente dürfen mit Markierungen (Tagged Values) ausgezeichnet werden. Eine Markierung besteht aus einem Namen und dem dazugehörigen Wert (Name/Value). Der Name ist z. B. „uid" und der Wert ist die generierte UID selbst.
Um eine Markierung programmatisch hinzufügen zu können, muss man sich zunächst das XMI Format genauer anschauen, das Poseidon verwendet:
Der Name einer Markierung wird zunächst als `<UML:TagDefinition>` Element definiert. TagDefinition ist ein direktes Kindelement von `<XMI:Content>` und damit im globalen Bereich definiert. `<UML:TagDefinition>` besitzt ein Attribut „name", in dem man den Namen angibt und ein Attribut „xmi.id", das für den Referenzierungsmechanismus innerhalb der XMI Datei gebraucht wird. Alle anderen Attribute oder Kindelemente spielen für die UID keine weitere Rolle.

UID Generierung

Die Grundvoraussetzung einer verwendbaren UID ist ihre Eindeutigkeit, d. h. jede neu erzeugte UID muss sich von allen bisher existierenden UIDs unterscheiden. Dies kann man noch auf einen Wirkungsbereich einschränken, für den die Eindeutigkeit gewährleistet sein muss. Eine UID lässt sich üblicherweise aus der Uhrzeit generieren und kann mit spezifischen Buchstabenkombinationen ergänzt werden.
Der IDGenerator verwendet die Klasse `java.rmi.server.UID` aus dem JDK 1.4.2. Objekte dieser Klasse repräsentieren je eine UID, die sich mit `UID.toString()` als String darstellen lassen. Eine solcherart generierte UID besteht aus drei Teilen

- Identifikation der VM, die sich aus Host-Namen und Uhrzeit zusammensetzt
- Zeit in Millisekunden als long Wert
- Wert des UID Zählers innerhalb der VM

Eine mögliche UID könnte folgendermaßen aussehen:

```
efd552:f913d92d0c:-7fe5
```

Das folgende Beispiel definiert den Namen „uid“ der Markierung:

```
<UML:TagDefinition xmi.id = 'lsm:ae1cf:f988cb67b2:-7f1e' isSpecification = 'false'
            name = 'uid'
            tagType = 'uid'>
      <UML:TagDefinition.multiplicity>
            <UML:Multiplicity xmi.id = 'lsm:ae1cf:f988cb67b2:-7f1f'>
                  <UML:Multiplicity.range>
                        <UML:MultiplicityRange
                              xmi.id = 'lsm:ae1cf:f988cb67b2:-7f20'
                              lower = '1'
                              upper = '1'/>
                        </UML:Multiplicity.range>
            </UML:Multiplicity>
      </UML:TagDefinition.multiplicity>
</UML:TagDefinition>
```

Listing 12: Definition von „uid“ in XMI

Nun muss der Wert der Markierung (die generierte UID) noch einem Modellierungselement zugeordnet werden. Dazu verwendet Poseidon das `<UML:ModelElement.taggedValue>` Element, das ein direktes Kindelement des Modellierungselements ist. Innerhalb des `<UML:ModelElement.taggedValue>` Elements gibt es das `<UML:TaggedValue.dataValue>` Element, das die UID speichert. Über das Attribut „xmi.idref“ des Elements `<UML:TagDefinition>` wird auf die Definition der Markierung Bezug genommen.

```
<UML:Class xmi.id = 'lsm:ae1cf:f988cb67b2:-7f85' name = 'GroupDealer'
            visibility = 'public'
            isSpecification = 'false'
            isRoot = 'false'
            isLeaf = 'false'
            isAbstract = 'false'
            isActive = 'false'>
      <UML:ModelElement.taggedValue>
            <UML:TaggedValue xmi.id = 'lsm:ae1cf:f988cb67b2:-7f98'
                  isSpecification = 'false'>
                  <UML:TaggedValue.dataValue>
                        16a786:f8f4003962:-7fde
                  </UML:TaggedValue.dataValue>
                  <UML:TaggedValue.type>
                        <UML:TagDefinition
                              xmi.idref = 'lsm:ae1cf:f988cb67b2:-7f1e'/>
                  </UML:TaggedValue.type>
            </UML:TaggedValue>
      </UML:ModelElement.taggedValue>
</UML:Class>
```

Listing 13: Die Definition der UID innerhalb eines TaggedValue in XMI

Design und Implementierung

Der IDGenerator besteht aus drei Klassen. Die Klasse IDGenerator enthält die `main()` Methode und steuert den Workflow. Die Markierung wird durch die Klassen TagDefinition und TaggedValue abgebildet. Dies ermöglicht dem IDGenerator das Arbeiten mit Objekten, die die etwas kompliziertere Struktur der XML Repräsentation kapseln. Eine Änderung der XMI Struktur lässt sich so auch leichter einpflegen.

Funktionsweise

Der IDGenerator liest die XMI Datei ein, die ihm als Kommandozeilen-Parameter übergeben wurde. Aus der XMI Datei wird eine XML DOM (Document Object Model) Repräsentation im Speicher aufgebaut. Der DOM Baum wird zuerst nach einer bereits vorhandenen TagDefinition „uid" abgesucht. Wurde eine gefunden, merkt sich der IDGenerator die xmi.id der TagDefinition, ansonsten legt er eine neues TagDefinition Element im DOM Baum an. Anschließend werden alle Modellierungselemente herausgefiltert, die mit einer UID versehen werden sollen. Jedes dieser Modellierungselemente wird dann überprüft, ob es bereits mit einer UID ausgezeichnet worden ist. Ist dies nicht der Fall, wird ein neues `<UML:ModelElement.taggedValue>` Element eingefügt, das eine neu generierte UID enthält.
Abschließend wird der DOM Baum wieder in die XMI Datei serialisiert.
Der IDGenerator wurde in der Diplomarbeit direkt in den Build Prozess eingebunden.

Der IDGenerator ist an die Interpretation des XMI Format von Poseidon gebunden, andere Werkzeuge exportieren möglicherweise in einer ähnlichen Struktur, so dass eine Anpassung des IDGenerators erfolgen muss.

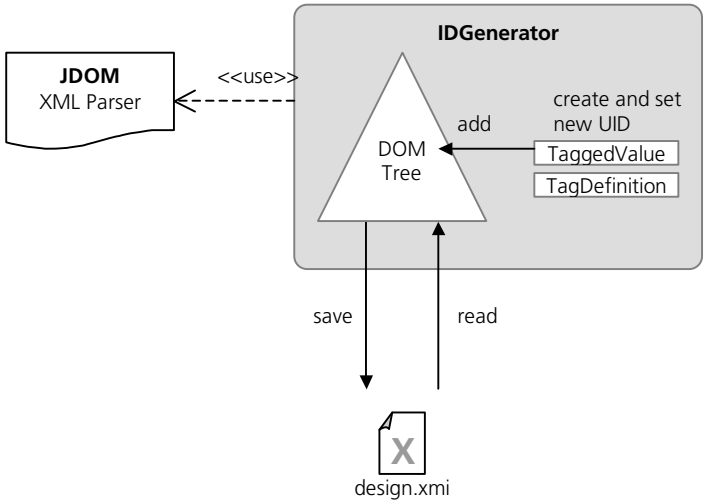

Abbildung 27: Funktionsweise des IDGenerators.

7.9 Build und Deployment Workflow

7.9.1 Build Prozess

Bevor aus einem PIM eine auslieferbare Software-Komponente für einen Web Application Server entsteht, müssen mehrere Prozesse ausgeführt worden sein:

- Modellierung des PIM: Das PIM einer Softwarearchitektur wird mit einem Werkzeug in UML nach der Vorgabe einer Designsprache modelliert.
- XMI Export: Das PIM wird mithilfe des XMI-Exports des UML Werkzeugs in einer XML Datei abgespeichert.
- UID Generierung: Der XMI Export wird von dem IDGenerator Utility eingelesen und eine bestimmte Auswahl von Modellierungselementen wird mit einer UID versehen. Die modifizierte Datei wird abgespeichert.
- Code-Generierung: Der b+m Generator wird mit dem XMI-Export und den Templates als Input gestartet. Der Generator generiert die Artefakte und speichert sie in einer Verzeichnisstruktur ab.
- Code-Formatierung: Die generierten Java-Dateien werden durch das Jalopy Code Formatter Utility formatiert.
- Kompilierung: Die generierten Java-Dateien werden mit einem Java-Compiler zu Byte-Code kompiliert.
- Paketierung: Die generierten Artefakte werde zunächst in JAR Archiven zusammengefasst und anschließend in ein EAR Archiv verpackt.
- Deployment: Das EAR Archiv wird in ein Verzeichnis des Application Servers ausgeliefert.

Für jeden dieser Prozesse können unterschiedliche Werkzeuge zum Einsatz kommen. Es wäre auf die Dauer sicher lästig und fehleranfällig, jeden Prozess einzeln von Hand anstarten zu müssen. Aus diesem Grund wird das Build-Prozess Werkzeug Ant verwendet. Ant erlaubt den Einsatz von Konfigurationsskripten, in denen man in einem XML Dialekt den Workflow von Prozessen beschreiben und externe Werkzeuge antriggern kann. Ein Prozess wird als „Target" definiert, der verschiedene „Tasks" zur Ausführung bringen kann. Zwischen einzelnen "Targets" können Abhängigkeiten festgelegt werden, so dass eine serielle Ausführung möglich ist.

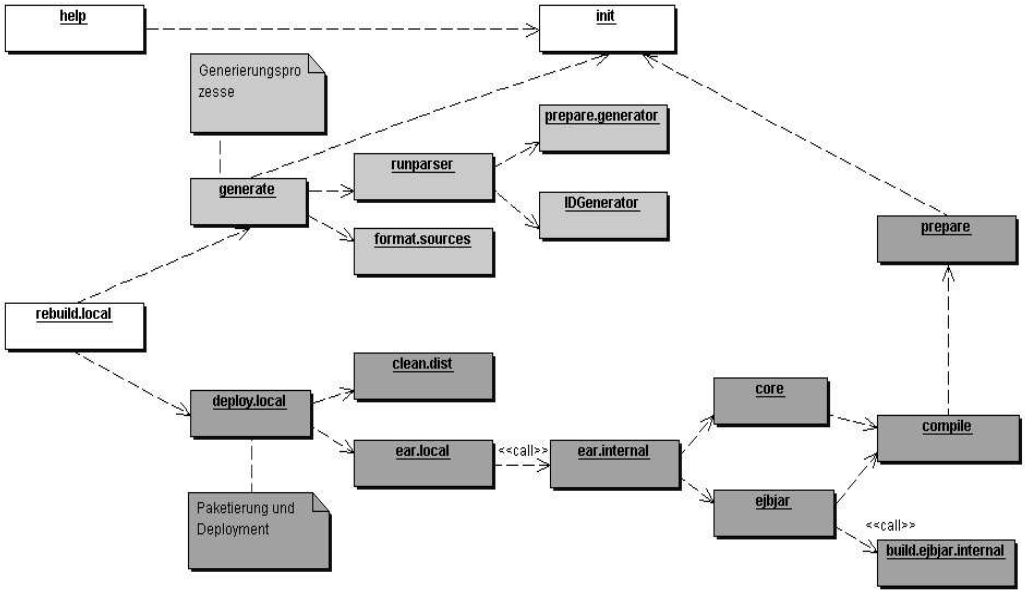

Abbildung 28: Der Graph mit den Abhängigkeiten der einzelnen Ant Targets

Die für den Generierungs- und Build-Prozess verwendeten Targets und ihre Abhängigkeiten sind in Abbildung 28 dargestellt. Die Ausführungsreihenfolge ergibt sich aus der Abhängigkeitskette und der Reihenfolge im Build-Skript.

Ant Targets

Die zentralen Targets werden im folgenden kurz vorgestellt. Eine detaillierte Beschreibung jedes Targets ist in den Kommentaren des Ant Build-Scripts zu finden.

generate startet den b+m Generator und übergibt den XMI-Export zusammen mit den Templates als Input Parameter. Die generierten Java Klassen werden anschließend mit Jalopy formatiert. Das Target sollte direkt gestartet werden, wenn man ausschließlich neu generieren möchte.

rebuild.local ist das zentrale Target, das den gesamten Generierungs- und Deployment Prozess ausführt.

deploy.local kompiliert die generierten Java Dateien, archiviert sie in JAR Archiven und verpackt diese in einem EAR Archiv, das an den Web Application Server ausgeliefert wird. Dieses Target sollte benutzt werden, wenn man ausschließlich den Deployment Prozess anstarten möchte, ohne neu zu generieren.

ear.internal ist ein internes Target, das von anderen Targets über einen "ant-call" aufgerufen werden kann. Das Target erzeugt das EAR Archiv, in das die JAR Archive core.jar und XXX.ejb.jar.jar, sowie weitere Biblitotheken (JUnit, Log4J, etc.) hineingepackt werden.

Konfiguration des Build-Prozesses

Variable Anteile in Ant Build Skripten können in externe Properties-Dateien ausgelagert werden und über Variablen innerhalb des Skripts angesprochen werden. Die properties Datei wird am Anfang des Build Skripts inkludiert.
Dies eignet sich beispielsweise besonders gut für Pfadnamen, die sich von Projekt zu Projekt unterscheiden können und nicht festkodiert in einem Build Skript notiert sein sollten.
Aus diesem Grund inkludiert das Ant Build Skript die Datei build.properties, in der u. a. die Pfadangaben für das Projekt, den Generator und den Application Server angegeben werden können. Ein neues Projekt erfordert dann in der Regel nur die Anpassung der Properties-Datei.

7.9.2 Paketierung

Das server-seitige Subsystem mit den Komponenten der Anwendung wird als EAR Archiv an den Application Server ausgeliefert. Das EAR Archiv enthält die JAR Archive, die hier näher beschrieben sind.

core.jar ist der Kern der Anwendung und enthält alle Nicht-EJB Komponenten wie Basis-, Exception-, Factory- und Utility-Klassen. Das Archiv soll für andere ähnliche Anwendungsarchitekturen wiederverwendbar und von den EJB Komponenten entkoppelt sein.

xxx.ejb.jar: Das Archiv enthält die EJB Komponenten (EnterpriseBeans, Interfaces) der Anwendung. xxx steht als Platzhalter für den Namen des Subsystems oder der Komponente.

junit.jar enthält die Klassen und Interfaces des JUnit Testframeworks. JUnit wird von dem Test-Client verwendet.

log4j_xxx.jar enthält das Logging Framework. xxx steht als Platzhalter für die Framework-Version.

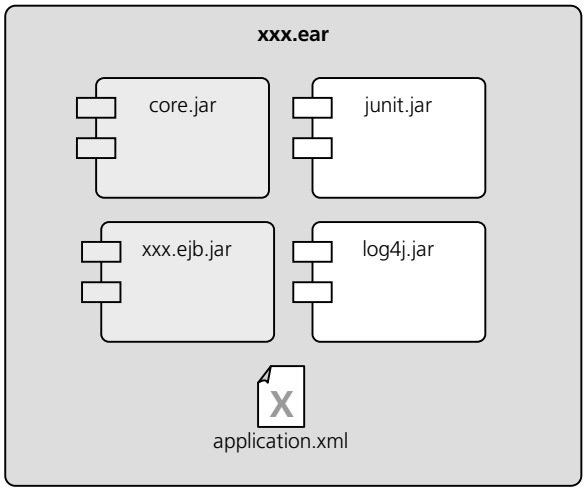

Abbildung 29: Das EAR Archiv

Verzeichnisstruktur:

xxx.ear
- /meta-inf
  - Manifest.ml
  - application.xml
- **xxx.ejb.jar**
  - /meta-inf
    - Manifest.ml
    - ejb-jar.xml
    - weblogic-ejb-jar.xml
    - weblogic-cmp-rdbms-jar.xml
  - ... (weitere Verzeichnisse)
- **core.jar**
- **junit.jar**
- ... (weitere Archive/Bibliotheken)

7.9.3 Deployment Deskriptoren

Für das Deployment exisitieren insgesamt vier Deployment Deskriptoren, wovon zwei (application.xml, ejb-jar.xml) durch die EJB Spezifikation vorgebenen sind und eine abstrakte Deklaration von EJB Komponenten erlauben, während die beiden anderen (weblogic-ejb-jar.xml, weblogic-cmp-rdmbs-jar.xml) rein technischer Natur und herstellerabhängig sind (hier von der BEA Weblogic Plattform).

application.xml

Der Application Descriptor wird durch http://java.sun.com/j2ee/dtds/application_1_2.dtd beschrieben und definiert den Zusammenbau einer Enterprise Application aus verschiedenen Modulen.

Beispiel:

```
<application>
      <display-name>IVSR Server</display-name>
      <description>DealerMasterdata Implementation</description>
      <module id="DealerMasterdata">
            <ejb>DealerMasterdata.ejb.jar</ejb>
      </module>
</application>
```

Listing 14: application.xml für DealerMasterdata

ejb-jar.xml

Der abstrakte Deployment Deskriptor wird durch http://java.sun.com/dtd/ejb-jar_2_0.dtd beschrieben. Er enthält drei wichtige Top-Level Container-Elemente, die für die Beschreibung der EJB Komponenten verwendet werden

`<enterprise-beans>`
In diesem Container werden die Enterprise Bean Komponenten deklariert.
Dies umfasst u. a. die EntityBeans

- logischer Name der EntityBean
- Local/Home Interface
- Local/Remote Interface
- Klassenname der EntityBean
- Persistenztyp (Container-managed, Bean-managed)
- Primary Key Klasse
- Feldnamen (Attributnamen der fachlichen Daten)
- EJB-QL Statements für Finder Methoden

und für SessionBeans

- logischer Name der SessionBean
- Local/Home Interface
- Local/Remote Interface
- Session-Typ (Stateless, Stateful)
- Transaktionstyp

`<relationships>`
In diesem Container Element werden die Beziehungen zwischen EntityBeans aus logischer Sicht beschrieben. Für jede Beziehung wird ein `<ejb-relation>` Element eingefügt, das folgende Informationen enthält:

- Name der Beziehung
- Die Rollen, aus denen sich eine Beziehung zusammensetzt. Die Rolle entspricht in UML einem Assoziationsende einer Klasse. Eine Beziehung (Assoziation) besteht immer aus genau zwei Rollen (Assoziationsenden). Pro Rolle wird ein Rollenname, Multiplizität (One, Many), die zugeordnete EntityBean und der Attributname der assoziierten EntityBean angegeben. Ist ein Attributname der jeweils assoziierten EntityBean in jeder der beiden Rollen definiert, handelt es sich um eine bidirektionale Beziehung, ansonsten ist sie unidirektional. Die Multiplizität einer Rolle hängt direkt mit der Attributdefinition der jeweils anderen Rolle zusammen. Wenn beispielsweise für eine Rolle `"Many"` als Wert für die Multiplizität angegeben wurde, muss in der anderen Rolle für das Attribut eine `java.util.Collection` oder ein `java.util.Set` definiert sein.

`<assembly-descriptor>`
In diesem optionalen Container Element können unter anderem die Zugriffsrechte für Methoden der Enterprise Beans vergeben werden.

weblogic-ejb-jar.xml

http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd beschreibt den technischen Deployment Deskriptor und erweitert ejb-jar.xml. Die einstellbaren Parameter sind spezifisch für die BEA Weblogic Plattform und umfassen Caching, Pooling, Clustering usw. Pro EnterpriseBean (<weblogic-enterprise-bean>) können der JNDI Name des Home Interfaces und der Deskriptor, der für das Mapping auf die Datenbank zuständig ist, angegeben werden.

weblogic-cmp-rdbms-jar.xml

http://www.bea.com/servers/wls810/dtd/weblogic-rdbms20-persistence-810.dtd beschreibt den technischen Deployment Deskriptor und legt das Mapping zwischen abstrakter und logischer Deklaration der EJB Komponenen in ejb-jar.xml und den physikalischen Gegebenheiten der Datenbank fest. Dies umfasst die Zuordnung von

- EntityBean Komponenten zu Tabellennamen
- Attributnamen zu Spaltennamen
- logischen Beziehungen zwischen EntityBeans zu technischen Beziehungen zwischen Tabellen

in zwei Top-Level Container Elementen.

-

`<weblogic-rdbms-bean>`
Hier werden die Eigenschaften der EntityBean auf die entsprechende Tabelle in der Datenbank gemapped.

`<weblogic-rdbms-relation>`
Eine Beziehung wird durch eine Primärschlüssel/Fremdschlüssel-Beziehung zwischen zwei Datenbanken ausgedrückt. Im Gegensatz zur abstrakten Deklaration in ejb-jar.xml wird für eine 1:1 oder 1:N Beziehung lediglich die Rolle angegeben, die der Tabelle mit den Fremdschlüssel-Spalten entspricht.

7.9.4 Application Server Deployment

Um eine paketierte und ausgelieferte EJB Anwendungskomponente in einem EJB Container ausführen zu können, muss der BEA Weblogic Application Server (WAS) entsprechend eingerichtet und konfiguriert werden.

Obwohl der WAS neue, in ein bestimmtes Verzeichnis ausgelieferte EAR Archive bei einem Neustart automatisch erkennen und laden kann, wird hier nochmals der manuelle Prozess dargestellt.

Nach dem Starten des WAS öffnet man einen Browser mit der URL `http://localhost:7001/console`. Damit gelangt man zum Login der Konfigurationskonsole. Nach dem erfolgreichen Login wird die WAS Home Seite geladen. Dort kann man unter der Rubrik „Domain Configurations“ den Hyperlink „Applications“ anklicken. Auf der „Applications“ Seite sieht man zunächst eine Liste von bereits registrierten Anwendungen (falls vorhanden) und man kann mit „Deploy a new Application“ eine neue Anwendung registrieren. Hier muss man lediglich den richtigen Pfad und das EAR Archiv der Anwendung auswählen und abschließend den „Deploy“ Button aktivieren.

Eine ausgeführte Registrierung wird dann entweder erfolgreich oder mit einer Fehlermeldung quittiert. Wenn man nun den WAS herunterfährt und erneut startet, wird die bereits registrierte Anwendung automatisch geladen.

8 Projektbeispiele

Der Einsatz des b+m Generator Frameworks (und damit der Templates und des Metamodells) wurde anhand zweier ausgewählter Softlab Software-Projekten getestet. Da die in den Projekten verwendeten Architekturen sehr komplex sind, wurde jeweils ein realisierbarer Ausschnitt als Vorlage für das Anwendungsdesign herausgenommen.
Es muss darauf hingewiesen werden, dass durch die bereits bestehenden Architekturen und Datenbankmodelle die Modellierung der PIMs eingeschränkt war. Entitäten und Anwendungsfälle waren vorgegeben. Es fand keine Modellierung auf der „grünen Wiese" statt, das bestehende Datenbankmodell diente als Vorlage.
Die folgenden Unterkapitel beschreiben die beiden Projekte zunächst in fachlicher Hinsicht und anschließend die Modellierung des PIM sowie den Generierungsprozess.

8.1 Aftersales Bonus System

Händler, die innerhalb Europas BMW Fahrzeuge vertreiben, erhalten einen Bonus, wenn sie bestimmte Rahmenbedingungen und Vorgaben der BMW AG erfüllen. Das Aftersales Bonus System erlaubt die Festlegung des Rahmens, in dem sich dieser Bonus bewegen kann, sowie die Berechnung und Anzeige des Bonus abhängig vom Erfüllungsgrad des jeweiligen Händlers.
Das System ist als dreischichtige Client/Server Architektur auf Grundlage der J2EE Plattform implementiert, mit einem Web-Client als Frontend und einem Application Server mit relationaler Datenbank im Backend. Für den Einsatz des b+m Generators wurde die Komponente „Bonusrahmen" ausgewählt [Raa01].

Es gibt reichlich redundante Code-Fragmente und Muster, die den Einsatz eines Code-Generators sinnvoll machen.

Zusätzlichen Nutzen kann man durch die Verwendung des MDA Ansatz errreichen:

- plattformunabhängige Modellierung
- übersichtliches Modell ohne technische Feinheiten
- Wiederverwendung von Templates und somit CA konforme Generierung des Architekturrahmens

Die Diplomarbeit versucht am Beispiel der Generierung der Business-Logik-, Datenzugriffs- und Persistenzschicht einer einzelnen Komponente dieses Systems die Möglichkeiten eines MDA konformen Generators aufzuzeigen. Auf die Generierung der Präsentationsschicht wurde hierbei aus Zeitgründen verzichtet.
Die Vorgehensweise an sich ist durch die bereits existierende Implementierung des Systems natürlich nicht typisch, sondern hat mehr den Charakter eines Reengineering unter dem Gesichtspunkt von MDA. So war für die Entwicklung des plattformunabhängigen Modells das bereits vorhandene Datenbankmodell ausschlaggebend. Im Optimalfall würde man aus dem PIM auch das Datenbankmodell bzw. die Datenbanktabellen generieren. Es ist jedoch durchaus realistisch, dass die Persistenzschicht vorgeben ist und man im Nachhinein Teile des fachlichen Modells ergänzt. Man muss also in gewisser Weise Modell und Persistenz abgleichen.

Um die generierte Architektur besser zu verstehen, werden zunächst die fachlichen Objekte der Komponente „Bonusrahmen“ und die Anwendungsfälle grob beschrieben. Eine detailliertere Beschreibung ist in [Ver01] zu finden, soll aber nicht Bestandteil der Diplomarbeit sein, da es hier um die Generierung des Architekturrahmens geht und nicht um die Implementierung der Fachlogik.

8.1.1 Entitäten

Es gibt verschiedene Arten von Boni, die einem Händler/Werkstatt zugeteilt werden können

- Standard (kein Bonus, also eine Art Null-Euro Bonus)
- Volumen- und Zielerreichungsbonus sind quantitative Boni, für die der Händler oder die Werkstatt bestimmte vorgegebene Umsätze und Ziele (z. B. Wachstumsziel) erreichen muss.
- CSI (Customer Satisfaction Index) Bonus und marktspezifischer Bonus sind qualitative Boni, bei denen sich der Händler oder die Werkstatt an verschiedene Qualitätsstandards halten und zur Kundenzufriedenheit beigetragen haben muss. Der Qualitätsstandard und die Kundenzufriedenheit werden von einem Auditor vor Ort überprüft.

Für die Speicherung der Auditorenergebnisse und sonstiger Berechnungen werden folgende Entitäten definiert:

Bonusrahmen: Der Bonusrahmen definiert

- das Verhältnis zwischen qualitativen und quantitativen Bonus
- das Verhältnis von Zielerreichungs- zu Volumenbonus
- das Jahr, für das der Rahmen festgelegt wird
- die Marke, für die der Rahmen festgelegt wird
- und den Mindesterfüllungsgrad der Retail Standards

Dem Bonusrahmen können ein Retail Standard Bonus und ein Bonusbaukasten pro Jahr zugeordnet werden.

Retail Standard Bonus: Um Anrecht auf einen Bonus zu haben, muss ein Händler zuerst einen gewissen Standard erfüllen, der durch die BMW AG vorgegeben wird. So ist beispielsweise die Anordnung und Nutzung von Präsentationselementen für den Neuwagenbereich in der Niederlassung standardisiert. Ein Retail Standard ist abhängig von der Bewertung eines Auditors erfüllt oder nicht erfüllt. Der Retail Standard Bonus besteht aus mehreren Retail Standards.

Bonusbaukasten: Der Bonusbaukasten ist ein Leistungsstandard und kann mehrere Bonuskomponenten enthalten, die einer bestimmten Kategorie zugeordnet sind: Prozess/Logistik, Marketing oder Service. Für jede dieser Komponenten gibt es ein Minimum an Bonusleistung.

Retail Standard Katalog: Der Retail Standard Katalog enthält einen Kurztext, der einem Retail Standard zugeordnet ist. Normalerweise würde man den Kurztext direkt als Attribut im Retail Standard abspeichern, da er jedoch extern befüllt wird, ist er in eine eigene Entität ausgelagert.

Text: Die Entität Text wird von den anderen Entitäten verwendet, um Beschreibungen abhängig von der Sprache des jeweiligen Händlers abzuspeichern.

8.1.2 Anwendungsfälle

Die Komponente erlaubt der Zentrale bei BMW AG, die Vorgaben eines jährlichen Bonusprogramms pro Marke festzulegen. Dieser Anwendungsfall lässt sich in drei weitere untergeordnete Anwendungsfälle aufteilen:

- *Allgemeinen Bonusrahmen festlegen*: Die Zentrale kann hiermit die allgemein gültigen Rahmenbedingungen für die Niederlassungen festlegen, die als Grundlage für die Bonusberechnung dienen.
- *Rahmen für Retail Standard Bonus festlegen*: Neben den allgemeinen Bedingungen können weitere Bedingungen für die Ausschüttung des Retail Standard Bonus festgelegt werden.
- *Baukasten festlegen*: Hiermit lässt sich der Inhalt des Baukasten anpassen. Einzelne Baukastenkomponenten können hinzugefügt oder gelöscht werden.

8.1.3 Projekt Konfiguration

Das Projekt wird als Eclipse Projekt unter dem Namen „VERA-Bonusrahmen“ in einem Verzeichnis desselben Namens im VSS Repository abgelegt. Dort werden drei Unterverzeichnisse /design, /config und /lib (vgl. Kapitel 7.3) erstellt.

Konfigurationsdateien

build.properties: Für die Anpassung des Ant Build Skripts an das Projekt muss lediglich ein Entrag verändert werden: `component.name=BonusRahmen`.
Alle anderen Einstellungen können übernommen werden.
map.xml: Die im PIM verwendeten Namen und Beziehungen der BusinessEntities müssen in der Datei auf die Gegebenheiten des Datenbankmodells abgebildet werden. Dies kann entweder manuell erfolgen oder durch Generierung eines Teils von map.xml aus dem PIM über das MapXml Template (vgl. Kapitel 7.4).
config.properties: Der allgemeine Pfad wird an das Projekt angepasst:
`common.path=de/softlab/vera`
Alle anderen Einstellungen können übernommen werden.

8.1.4 Modellierung des plattformunabhängigen Modells für das Backend

Die Anwendungsfälle und fachlichen Objekte müssen nun mithilfe des UML CASE Werkzeugs Poseidon in ein Klassendiagramm übertragen werden, das die Anwendungsarchitektur und damit das PIM darstellt. Das grundlegende Design für das PIM wird durch die BMW Component Architecture und das bereits existierende Datenbankmodell vorgegeben. Das Datenbankmodell ist technischer Natur und wird der Einfacheit halber im PIM weitgehend identisch in Form von BusinessEntities abgebildet.

BusinessComponent

Als erster Schritt wird die Komponente „Bonusrahmen“ in das Diagramm gezeichnet. Alle weiteren Klassen sind Bestandteil dieser Komponente. Die nächsten Schritte ergeben sich direkt aus der CA.

BusinessFacade

- ***BonusRahmen***: Die BusinessFacade stellt die Schnittstelle der Komponente nach aussen zur Verfügung. Sie enthält die Anwendungsfälle als Methodensignaturen. Die BusinessFacade ist abhängig von den unten erwähnten BusinessActivities. Das Modell

wurde so gewählt, dass es pro Anwendungsfall eine eigene BusinessActivity gibt, in der sich die Fachlogik befindet.

BusinessActivities

- ***InsertBonusRahmen:*** Fügt einen neuen Bonusrahmen hinzu und ist damit abhängig von BusinessEntity BonusRahmen.
- ***SaveBonusRahmen:*** Speichert modifzierte Änderungen eines Bonusrahmens und ist damit abhängig von BusinessEntity BonusRahmen.
- ***ReleaseBonusRahmen:*** Gibt einen Bonusrahmen frei und ist damit abhängig von BusinessEntity BonusRahmen.
- ***VerifyBonusRahmen:*** Validiert einen Bonusrahmen und ist damit abhängig von BusinessEntity BonusRahmen.
- ***GetBonusRahmen:*** Liefert einen bestimmten Bonusrahmen zurück und ist damit abhängig von BusinessEntity BonusRahmen.
- ***GetSelectionRetailStandards:*** Liefert eine bestimmte Auswahl an Retail Standards zurück und ist damit abhängig von BusinessEntity RetailStandard.
- ***DeleteRetailsStandards:*** Löscht einen bestimmten Retail Standard und ist damit abhängig von BusinessEntity RetailStandard
- ***DeleteBonusKomponente:*** Löscht eine Bonuskomponente aus dem Baukausten und ist damit abhängig von BusinessEntity Baukasten
-

BusinessEntities

- ***BonusRahmen*** ist die zentrale Entität der Komponente und verfügt über eine 1:N Beziehung zu RetailStandard und Baukasten und 1:1 Beziehungen zu Text und RetailStandardKatalog, die über die Rollennamen „volumenText“, „csiText“, „baukasten“, „retailStandard“ und „retailStandardKatalog“ identifiziert werden. Die Attribute jahr und vertriebsSchieneID dienen als eindeutiger Primärschlüssel.
- ***RetailStandard*** enthält die Attribute jahr, vertriebsSchieneID und Retail Standard ID, die als eindeutiger Primärschlüssel dienen.
- ***Baukasten*** verfügt über zwei 1:1 Beziehungen zu Text, die durch die Rollennamen "name" und "beschreibung" identifiziert werden. Die Attribute baukastenID, Jahr und vertriebsschieneID dienen als Primärschlüssel.
- ***RetailStandardKatalog*** verfügt über eine 1:N Beziehung zu RetailStandard, die durch den Rollennamen "retailStandard" identifiziert wird. Die Attribute retialStandardID, und vertriebsschieneID dienen als Primärschlüssel.
- ***Text*** wird durch das Attribut textID identifiziert.

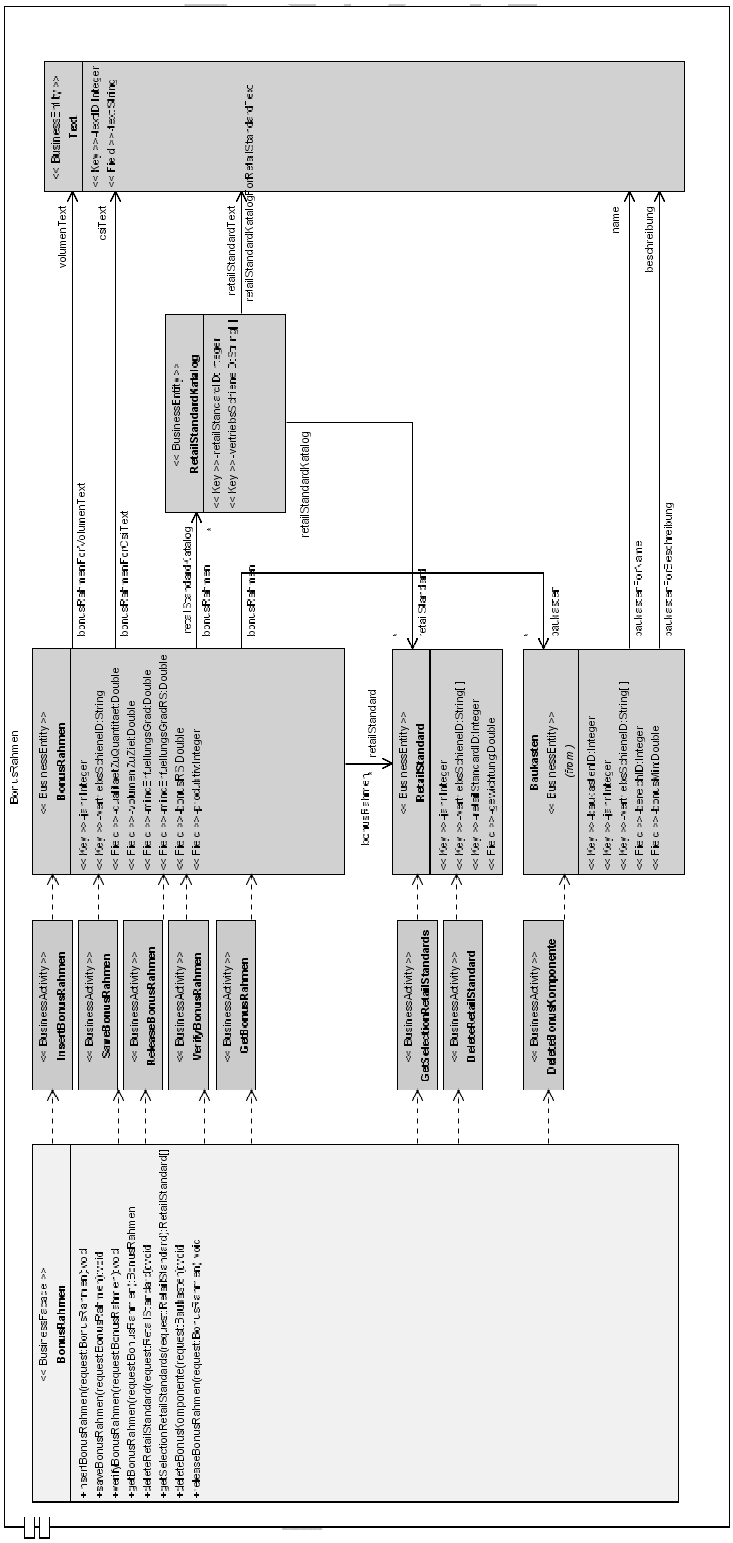

Abbildung 30: Klassendiagramm des PIMs „Bonusrahmen“

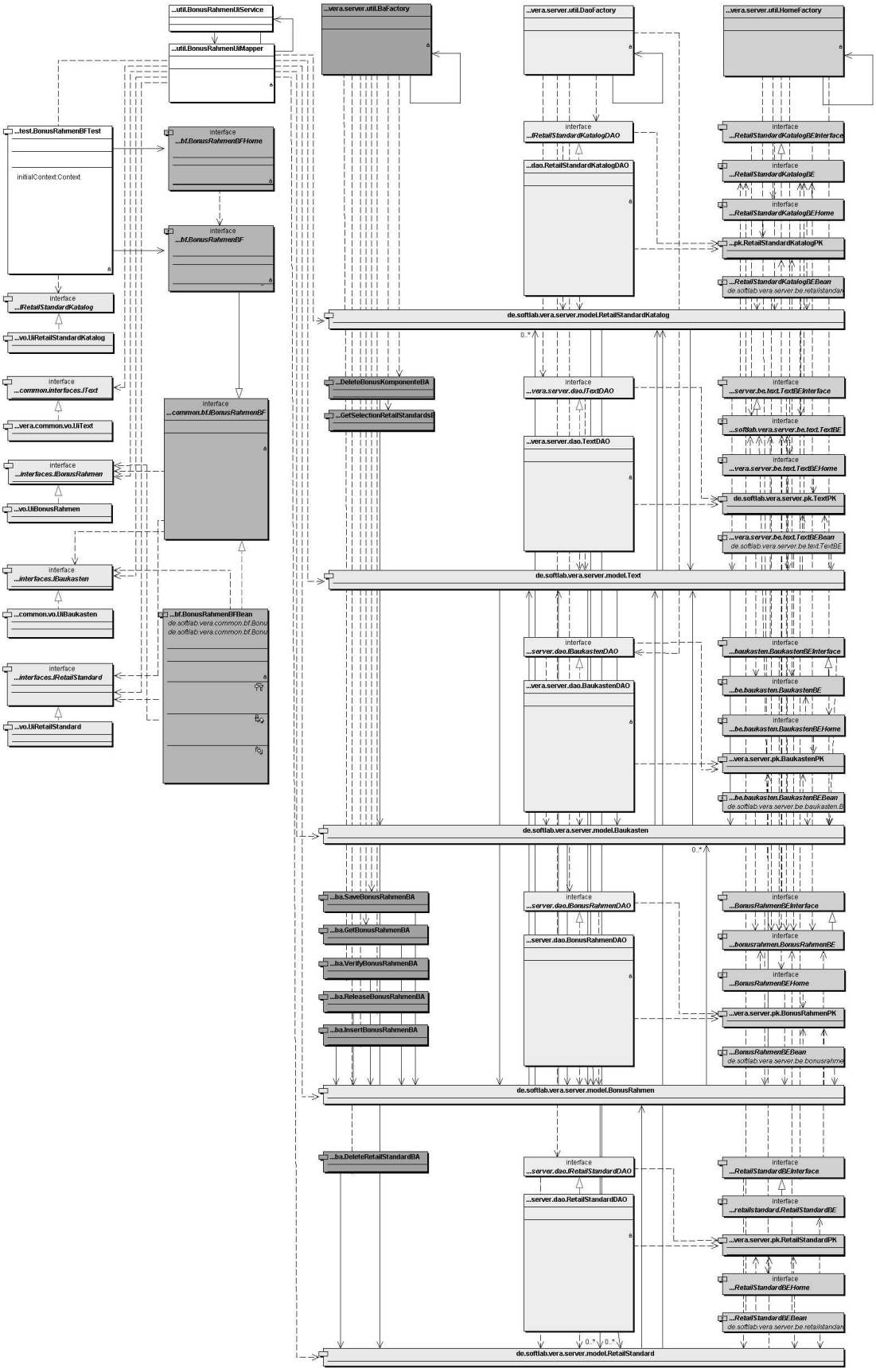

Abbildung 31: Generierter Architekturrahmen „Bonusrahmen“.

8.1.5 Ergebnis

Das modellierte Anwendungsdesign wird mit Poseidon nach XMI exportiert und im /design Verzeichnis abgespeichert. Der Generierungsprozess wird über das „generate“ Target des Ant Build Skripts angetriggert und führt zu dem in Abbildung 31 dargestellten Ergebnis.

Wie schön zu erkennen ist, wird eine Vielzahl an Klassen generiert, die ausschließlich die technische Infrastruktur abbilden. So entstehen beispielsweise aus jeder mit dem Stereotyp <<BusinessEntity>> ausgezeichneten Klasse im PIM fünf EJB-spezifische Klassen/Interfaces, ein Model, ein Value Object und ein Data Access Object. Jede dieser Klassen wird vollständig generiert und benötigt keinerlei manuellen Ergänzungen.
Man kann sich leicht vorstellen, dass eine Implementierung dieser Klassen zu „Fuß“ erhebliche Zeit in Anspruch genommen hätte und fehlerträchtig gewesen wäre.

8.2 Händler-Stammdatenverwaltung

Einen besonderen Datenbestand in den Unternehmen bilden die sogenannten Stammdaten, Daten, die die Kerninformationen des Unternehmens ausdrücken. Diese Daten können z. B. aus den Adressen und ergänzenden Informationen über den Kundenstamm oder die Vertragshändler des Unternehmens bestehen. Darauf aufbauend können diese Daten von weiteren abgeleiteten Informationsystemen anderer Domänen verarbeitet und in domänenspezifischem Kontext wiederverwendet werden.

Stammdaten sind Daten, welche über längere Zeit nicht verändert und bei der Abwicklung von Geschäftsvorfällen verwendet werden.

Beispiele:

- Produktspezifikationen (Materialstämme, Stücklisten, etc.)
- Kaufmännische Konditionen
- Kunden- und Lieferantenangaben

Bei der BMW AG besteht der Stammdatenbestand z. B. aus Informationen über die Struktur und die Adressen des Vertragshändlernetzes.
Die Pflege dieser Stammdaten ist eine Teilfunktionalität des IVS-R (International Vehicle System – Reengineering) Projekts, das Softlab GmbH zusammen mit sd&m für die BMW AG entwickelt hat.
Es handelt sich dabei um eine dreischichtige Client/Server Architektur auf Basis von Java 2 Enterprise Edition (J2EE). Für die Persistenzschicht werden Enterprise Java Beans über Application Server verwendet. Auf dem Client kommt JSP zur Anwendung [Ban01].

Die relativ simplen und wiederkehrenden Anwendungsfälle [Lau01], [Lau02] wie das Anlegen, Lesen und Löschen von Stammdaten und die Anzahl der Datenbanktabellen sprechen für den Einsatz eines Generators, um die EJB Komponenten zu erzeugen.

8.2.1 Entitäten

Die Stammdaten bestehen aus etwa dreissig Entitäten, wovon zwei als Beispiel für die Generierung ausgesucht wurden:

GroupDealer: Der GroupDealer ist der marktspezifische Auftraggeber an die BMW für Neufahrzeuge im Grosshandel. Der GroupDealer ist exakt einem Wholesaler (eine zentrale Vertriebseinheit) zugeordnet. Ihm sind mehrere DomesticDealer untergeordnet.
DomesticDealer: Der DomesticDealer repräsentiert einen regionalen BMW Händler, der einem GroupDealer und gleichzeitig auch einem Wholesaler (durch wholesalerID, wholesalerType) zugeordnet ist.

8.2.2 Anwendungsfälle

Für die Verwaltung aller Entitäten der Händler-Stammdaten sind sehr simple Anwendungsfälle vorgesehen, die eine Kernfunktionalität für die Wartung einzelner Attribute bereitstellen:

Der Anwendungsfall Maintain (Wartung des Händlerstamms) steht stellvertretend für Anwendungsfälle aller weiteren Entitäten und setzt sich aus folgenden Sub-Anwendungsfällen zusammen:

- *Add* – Fügt einen neuen Eintrag hinzu, z. B. Add GroupDealer
- *Modify* – Ändert einen bereits bestehenden Eintrag, z. B. Modify GroupDealer
- *Delete* – Löscht einen Eintrag, z. B. Delete GroupDealer
- *Get* – Lädt einen Eintrag, z. B. Get GroupDealer
- *List* – Lädt eine Liste von Einträgen, z. B. List GroupDealer

8.2.3 Projekt Konfiguration

Das Projekt wird als Eclipse Projekt unter dem Namen "IVSR-DealerMaterdata" in einem Verzeichnis desselben Namens im VSS Repository angelegt. Dort werden drei Unterverzeichnisse /design, /config und /lib (vgl. Kapitel 7.3) erstellt.

Konfigurationsdateien

build.properties: Für die Anpassung des Ant Build Skripts an das Projekt muss lediglich ein Entrag verändert werden: `component.name=DealerMasterdata`.
Alle anderen Einstellungen können übernommen werden.
map.xml: Wie schon bei beim Projektbeispiel VERA Bonusrahmen kann das Mapping entweder manuell erfolgen oder durch Generierung eines Teils von map.xml aus dem PIM über das MapXml Template (vgl. Kapitel7.4).
config.properties: Der allgemeine Pfad wird an das Projekt angepasst:
`common.path=de/softlab/ivsr`
Alle anderen Einstellungen können übernommen werden.

8.2.4 Modellierung des plattformunabhängigen Modells für das Backend

Analog zum Projektbeispiel Aftersales Bonus System wird das PIM mit Poseidon UML gezeichnet.

BusinessComponent

Als erster Schritt wird die Komponente "DealerMasterdata" in das Diagramm gezeichnet. Alle weiteren Klassen sind Bestandteil dieser Komponente. Die nächsten Schritte ergeben sich direkt aus der CA.

BusinessFacade

- ***DealerMasterdata***: Die BusinessFacade stellt die Schnittstelle der Komponente nach aussen dar. Sie enthält die Anwendungsfälle als Methodensignaturen. Die BusinessFacade ist abhängig von den unten erwähnten BusinessActivities. Das Modell wurde so gewählt, dass es pro Anwendungsfall eine eigene BusinessActivity gibt, in der sich die Fachlogik befindet.

BusinessActivities

Die folgenden Activities stehen beispielhaft für alle weiteren Activities der anderen BusinessEntities:

- ***AddGroupDealer:*** Fügt einen neuen GroupDealer hinzu und ist damit abhängig von BusinessEntity GroupDealer.

- ***ModifyGroupDealer:*** Speichert die aktuellen Werte eines GroupDealers ab und ist damit abhängig von BusinessEntity GroupDealer.
- ***DeleteGroupDealer:*** Löscht einen bestimmten GroupDealer aus der Datenbank und ist damit abhängig von BusinessEntity GroupDealer.
- ***GetGroupDealer:*** Liefert einen bestimmten GroupDealer zurück und ist damit abhängig von BusinessEntity GroupDealer.
- ***ListGroupDealer:*** Liefert alle eingetragenen GroupDealer zurück und ist damit abhängig von BusinessEntity GroupDealer.
- ***ListDescription:*** Liefert alle Beschreibungen aller GroupDealer zurück und ist damit abhängig von BusinessEntity Description.
- …
-

BusinessEntities

- ***GroupDealer*** ist die zentrale Entität der Komponente und verfügt über eine 1:N Beziehung zu DomesticDealer und eine 1:1 Beziehung zu Description, die über die Rollennamen „domesticDealer“ und „description“ identifiziert werden. Die Attribute groupDealerID und groupDealerType dienen als eindeutiger Primärschlüssel.
- ***DomesticDealer*** enthält die Attribute wholesalerID, wholesalerType und ordererDomestic, die als eindeutiger Primärschlüssel dienen.
- ***Description:*** Das Attribut descriptionID dient als Primärschlüssel.

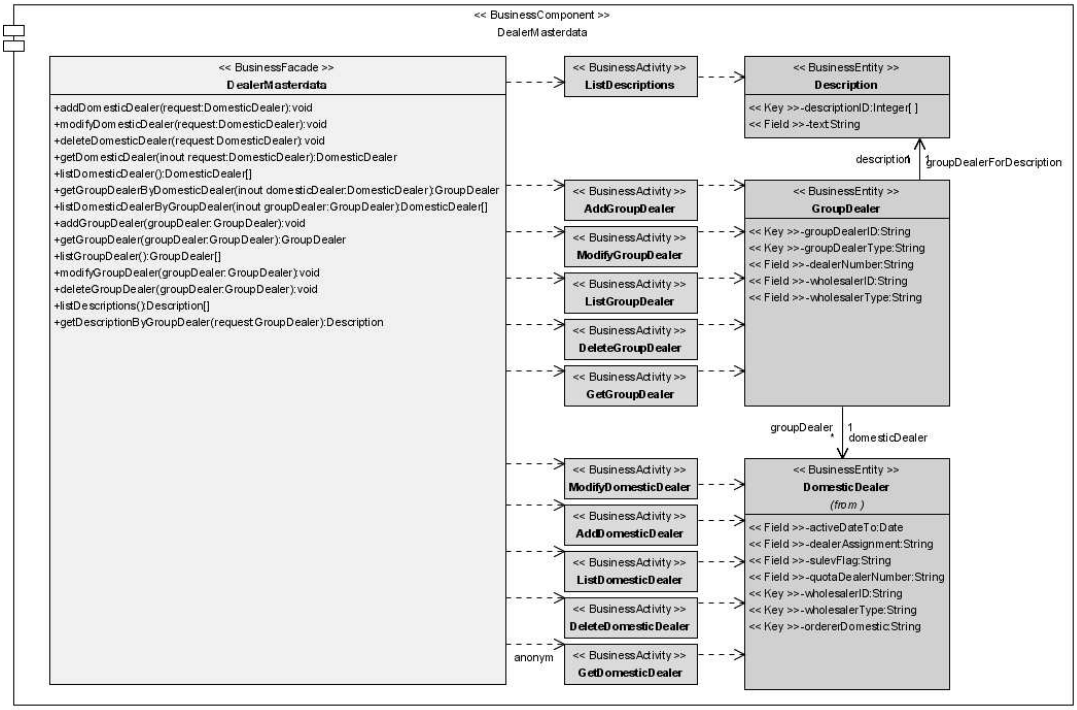

Abbildung 32: Klassendiagramm des PIMs „DealerMasterdata“

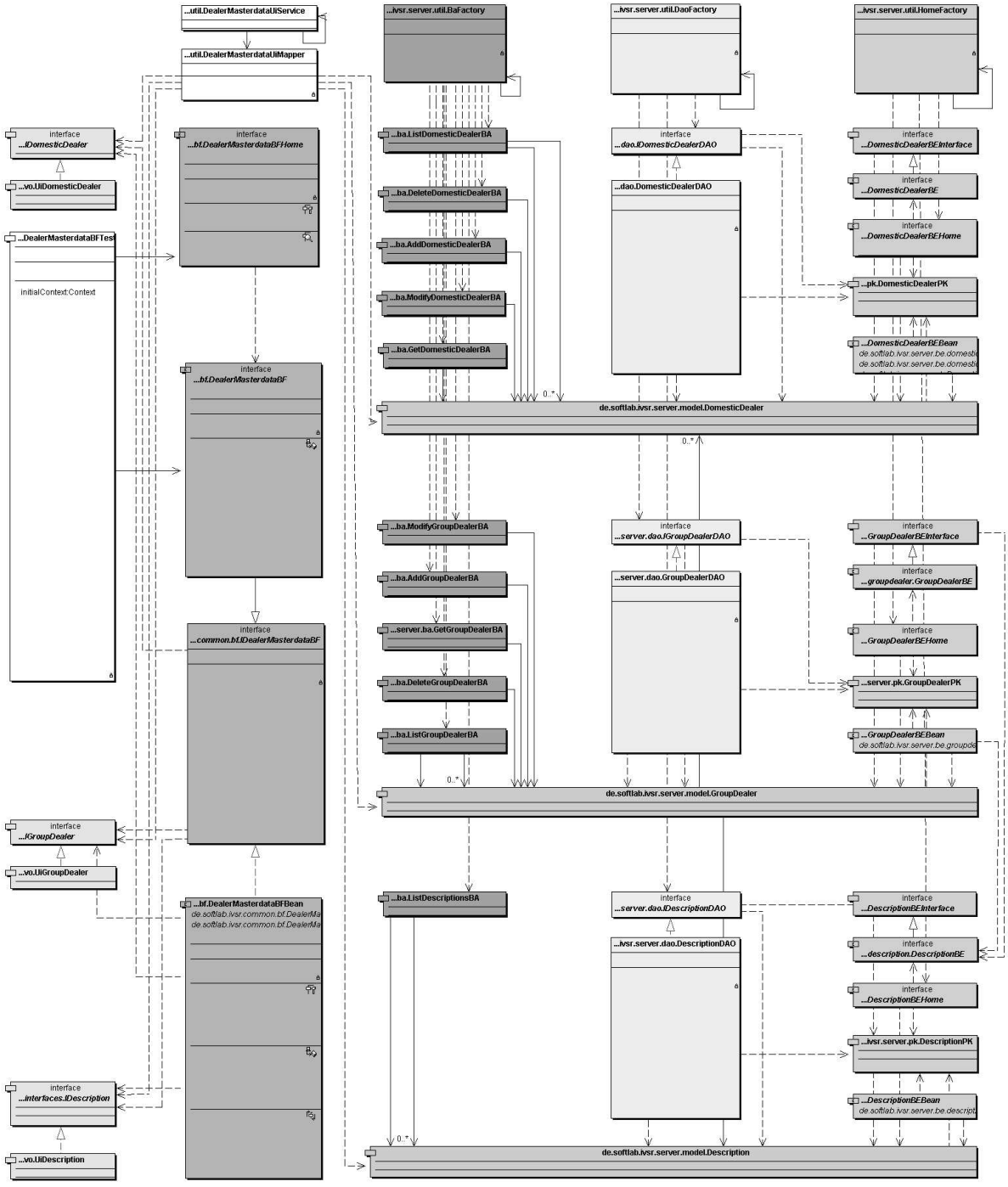

Abbildung 33: Generierter Architekturrahmen „DealerMasterdata“

8.2.5 Ergebnis

Das modellierte Anwendungsdesign wird mit Poseidon nach XMI exportiert und im /design Verzeichnis abgespeichert. Der Generierungsprozess wird über das „generate" Target des Ant Build Skripts angetriggert und führt zu dem in Abbildung 33 dargestellten Ergebnis.

Wie schon beim Aftersales Bonus System (vgl. Kapitel 8.1) wird eine Vielzahl an Klassen der technischen Infrastruktur generiert. Wenn man die Architekturrahmen beider Projekte vergleicht, erkennt man in der Struktur des Klassendiagramms die Architekturvorgabe, die auch durch die Anordnung der Klassen betont wird.

Der Architekturrahmen wurde anschließend mit Fachlogik ergänzt und als EAR Archiv an den Application Server ausgeliefert. Ein Test Case, der die Methoden der Session Facade ausführt, liefert folgenden Konsolen-Output zurück:

```
== AddGroupDealer() ==================================
|groupDealerID:462112|groupDealerType:00|dealerNumber:90982|wholesalerID:000004|wh
olesalerType:00| added.

== ListDescriptions() ================================
|descriptionID:6|text:dummy|
|descriptionID:2|text:dummy|
|descriptionID:1|text:dummy|
|descriptionID:7|text:dummy|
|descriptionID:8|text:dummy|

== GetDescriptionByGroupDealer() =====================================
|descriptionID:8|text:dummy|

== GetGroupDealer() ==================================
|groupDealerID:032900|groupDealerType:00|dealerNumber:null|wholesalerID:null|whole
salerType:null|

== ListGroupDealer() =================================
|groupDealerID:462112|groupDealerType:00|dealerNumber:90982|wholesalerID:000004|wh
olesalerType:00|

== ModifyGroupDealer() ===============================
|groupDealerID:462112|groupDealerType:00|dealerNumber:70349|wholesalerID:000004|wh
olesalerType:00| modified.

== DeleteGroupDealer() ===============================
|groupDealerID:462112|groupDealerType:00|dealerNumber:70349|wholesalerID:000004|wh
olesalerType:00| deleted.
```

Listing 15: Konsolen-Output des Test Cases (Auszug).

9 Rückblick

Zu Beginn meiner Diplomarbeit lag die wesentliche Aufgabenstellung in der Generierung von Quellcode, um „langweilige“ und wiederkehrende Aufgaben bei der Softwareentwicklung wie beispielsweise den Programmcode der technischen Infrastruktur von EJB Komponenten nicht von Hand ausführen zu müssen. Es war zunächst nicht entschieden, welcher Ansatz für einen Generierungsprozess genutzt werden sollte.

Sicherlich trug das derzeitige und bis heute andauernde große Interesse der Industrie an MDA, sowie Artikel in einschlägigen Fachzeitschriften dazu bei, den MDA Ansatz näher zu untersuchen und anhand von praktischen Beispielen aus Projekten auszuführen.
Da die Architektur der Händler-Stammdatenverwaltung des IVS-R Projekts als Grundlage der Diplomarbeit dienen sollte, verbrachte ich die ersten Wochen zunächst damit, eine komplette Laufzeitumgebung zu installieren, um die Anwendung zur Ausführung bringen zu können. Dazu war die Einarbeitung in die EJB Spezifikation und die Dokumentation des BEA Weblogic Application Server notwendig.

Nachdem ich mich nach Absprache mit meinem Betreuer dazu entschieden hatte, den MDA Ansatz weiter zu verfolgen, stellte sich die Frage, ob man einen MDA konformen Generator selbst enwickelt oder auf bereits bestehende Werkzeuge oder Frameworks zurückgreift. Nach einigen Recherchen kamen wir zu dem Schluss, dass die Erweiterung und Anpassung des b+m Generator Frameworks der b+m Informatik AG die beste Lösung ist, zu einem schnellen und effektiven Ergebnis zu gelangen. Eine Eigenentwicklung hätte unnötigerweise das Rad neu erfunden und wäre zeitlich kaum machbar gewesen.

Durch die Wahl des b+m Generator Frameworks ergab sich die Möglichkeit, sich beim weiteren Vorgehen an dem von der b+m Informatik AG entwickelten Generative Development Process (GDP) zu orientieren, einer pragmatischen Interpretation der MDA. GDP schlägt die Erstellung einer Referenzimplementierung anhand einer konkreten Anwendung vor, die dann als Grundlage für die Templates des Generators dient.
Aus diesem Grund wurde der bestehende Quellcode der Händler-Stammdaten Anwendung auf die wesentlichen Bestandteile reduziert (Ausführung der Anwendungsfälle ohne Validierung und Security), um unnötige Komplexität für die Erstellung der Templates zu vermeiden. Gleichzeitig erfolgte eine Anpassung an die Spezifikation der BMW Component Architecture (CA).

Für die Verwendung des b+m Generator Frameworks musste ich mich in das Prinzip der Metamodelle und die proprietäre Template-Sprache XPand einarbeiten. Ersteres war möglich durch die bei der OMG erhältlichen Spezifikationen zu UML und MOF und nicht zu vergessen den Inhalten der von mir besuchten Informatik 4 Vorlesung meines Studiengangs. Auf der Web Site der b+m Informatik AG waren weitere Dokumentationen zu finden, sowie ein Diskussionsforum auf den SourceForge.net-Seiten [b+mGenFw] des b+m Generator Frameworks.
Die Erweiterung des Metamodells ergab sich aus den Vorgaben der CA, die die Designsprache und damit die Auszeichnung der PIMs mit Stereotypen und Design Constraints ermöglichte. Die Erstellung der Templates aus der Referenzimplementierung nahm aufgrund der

Einarbeitung in eine neue Skriptsprache und die mangelnde Editorunterstützung einen beträchtlichen Zeitaufwand ein.
Die Modellierung des PIMs und der Einsatz eines MDA konformen Generators erforderte ein UML CASE Werkzeug, das eine XMI Export Schnittstelle besitzt, die der Generator versteht. Da sich der XMI Export des durch die BMW AG vorgegeben UML Werkzeug Together als problematisch erwies, habe ich mich für eine frei erhältliche Version von Poseidon UML entschieden.

Die Fertigstellung der Templates, des Metamodells und der Modellierung des PIM aus der Referenzimplemtierung der Händler-Stammdaten Verwaltung ermöglichte den erstmaligen Einsatz des b+m Generators. Durch Kompilierung des Quellcodes und testweises Deployment an den Application Server konnten Fehler in den Templates und im Metamodell entfernt werden. Nach der Ergänzung des generierten Architekturrahmens durch die Fachlogik der Anwendungsfälle in den geschützten Bereichen war über einen Testcase ein Durchstich zur Datenbank möglich und damit ein erstes wichtiges Ziel erreicht.

Gegen Ende der Diplomarbeit bestand ein weiteres Ziel in der Anwendung der Templates auf das Subsystem bzw. eine Komponente eines anderen Projekts, um damit einen Beweis für die Wiederverwendbarkeit zu erbringen. Dies erforderte eine Flexibilisierung der Projektstruktur und die Einführung einer Konfigurationsmöglichkeit des Build und Deployment Skripts.

Als weiteres Anwendungsbeispiel wurde die Komponente „Bonusrahmen“ des Aftersales Bonus Systems (vgl. Kapitel 8.1) ausgewählt. Wie schon bei der Händler-Stammdatenverwaltung war ein Datenmodell vorhanden, das als Ausgangspunkt für die Modellierung des PIMs diente. Der Generierungsprozess konnte ebenfalls erfolgreich ausgeführt werden, es war jedoch aus Zeitgründen leider nicht mehr möglich, die Fachlogik der Anwendungsfälle zu ergänzen und ein Deployment auszuführen.

9.1 Fallstricke

Der Einsatz von MDA und MDA konformen Werkzeugen wirft nicht unerhebliche Probleme und Schwierigkeiten auf, auf die ich in den folgenden Punkten näher eingehen möchte.

XMI Format

XMI ist ein von der OMG spezifiziertes Austauschformat für UML konforme Modelle, um die Interoperabilität zwischen verschiedenen UML Werkzeugen und auch MDA Generatoren zu ermöglichen. Gerade zu Anfang meiner Recherchen und Tests hat sich für mich herausgestellt, dass XMI nicht gleich XMI ist. Jeder Hersteller eines CASE Werkzeugs scheint so seine eigene Interpretation der XMI Spezifikation zu haben. Oft sind es marginale Unterschiede, die einen Austausch verhindern. So werden beispielsweise von Together manche Eigenschaften von Modellierungselementen als Tags exportiert, während Poseidon stattdessen Attributwerte verwendet.
Die Inkompatibilät führt zu einer Bindung an ein bestimmtes Werkzeug. Im schlechtesten Fall können bei einem Wechsel zu einem anderen UML Werkzeug bereits vorhandene Diagramme nicht übernommen werden. Es ist zu hoffen, dass sich die Hersteller in Zukunft enger an die XMI Spezifikation halten oder verschiedene XMI Export und Import Schnittstellen anbieten.

Das b+m Generator Framework reagiert darauf mit einem Adapter-Mechanismus, um sich von den verschiedenen XMI Export Formaten der Hersteller unabhängig zu machen.

Templates

Templates setzen sich aus einem statischen und einem dynamischen Teil zusammen. Beide Teile bestehen üblicherweise aus zwei verschiedenen Sprachen, z. B. XPand und Java oder XML, für deren Validierung zur Laufzeit herkömmliche Editoren keine Unterstützung bieten, was zu einer erheblichen Fehlerquote in den Templates führen kann. Für XPand gab es keine oder nur geringe Unterstützung durch ein Eclipse-Plugin, welches wenigstens Syntax-Highlighting oder Code-Completion zur Verfügung stellte. Dies macht die Editierung sehr zeitaufwändig und Fehler können erst zur Generierungs- oder Kompilierungszeit festgestellt werden.

Nach meinen Recherchen verwenden die momentan erhältlichen MDA konformen Werkzeuge und Frameworks jeweils unterschiedliche Skriptssprachen für die Definition von Templates. So verwendet AndroMDA Velocity, ArcStyler JPython. Dadurch ist man an einen Hersteller gebunden und Templates müssen komplett neu geschrieben werden, sobald es zu einem Wechsel des Generators kommt.

Metamodell

Metamodelle sind ein zentraler Bestandteil von MDA zur Beschreibung von PIMs, ohne deren Verständnis ein MDA konformer Generator nicht zum Laufen zu bringen ist. Durch sie entsteht eine zusätzliche Komplexität während des Entwicklungsprozesses. Entwickler müssen sich nicht nur mit einer Template-Sprache, sondern auch mit dem nicht leicht verständlichen Prinzip der Metamodelle auseinandersetzen.

Um Metamodelle erweitern zu können, müssen sie in einer konkreten Implementierung in einer Programmiersprache z. B. Java vorliegen. Durch die Programmiersprache hat man ein mächtiges Werkzeug an der Hand, um den Transformationsprozess zu steuern und bespielsweise externe Informationen einzulesen, was insgesamt ein äusserst positiver Punkt ist.

Da konkrete Implementierungen von Metamodellen von Generator zu Generator unterschiedlich sein können, ist man möglicherweise wiederum an das Werkzeug gebunden. Die Austauschbarkeit von Metamodellimplementierungen des b+m Generator Frameworks sind von mir nicht näher untersucht worden.

O/R Mapping

Das Mapping beschreibt die Abbildung einer logischen und objektorientierten Sicht auf die physikalischen Gegebenheiten der Datenbank. Dies macht zum einen die Anreicherung des Transformationsprozesses mit externen technischen Daten über das Metamodell erforderlich. Oft sind die Datenbankstrukturen bereits vorhanden und man muss sich Gedanken machen, wie beispielsweise vorhandene Tabellen und deren Beziehungen untereinander in ein technisch unabhängiges Modell passen.

Es hat sich gezeigt, dass eine bis auf einige Anpassungen direkte Abbildung von Tabellen in BusinessEntities des PIM am einfachsten zu realisieren ist, aber von der reinen Lehre abweicht. Nach [Sha03] beschreibt der abstrakte Deployment Deskriptor ejb-jar.xml die logische Sicht und die technischen Deployment Deskriptoren des EJB Containers die Umsetzung in die technische Sicht der Persistenzschicht. Bis jetzt ist für mich offen, inwieweit man den Übergang von logischer in die technische Sicht des Datenbankmodells automatisiert in den Deployment Deskriptoren generieren kann.

Konsistenz

Wenn Entwickler nach der Generierung der Artefakte ausserhalb der geschützten Bereiche Code oder weitere Klassen hinzufügen, die eigentlich generierbar wären, ändern sie damit möglicherweise die Semantik der Architektur. Werden diese Änderungen nicht manuell im Design nachgezogen, ist die Konsistenz und Synchronität zwischen Modell und Quellcode gefährdet und beide entwickeln sich auseinander. Dass dies schnell passieren kann, ist durchaus realistisch. Hier ist eine vorausschauende Planung des Software-Architekten oder Designers und die Disziplin der Anwendungsentwickler notwendig.

Geschützte Bereiche

Geschützte Bereiche in den generierten Artefakten müssen so flexibel definiert sein, dass ein Entwickler Fachlogik hinzufügen kann, ohne in nichtgeschützte Teile hineinschreiben zu müssen. So ist es nicht ausreichend, lediglich bestimmte Methodenkörper zu schützen, sondern auch Bereiche im Header oder innerhalb des Klassen-Scopes. So muss es für den Entwickler möglich sein, neue Attribute, Methoden und `import` Anweisungen einzufügen.
Es hat sich herausgestellt, dass Editoren, die den Entwickler bei der Programmierung automatisch unterstützen (wie z. B. der Java-Editor in Eclipse), kontraproduktiv sein können, da sie die besondere Semantik der geschützten Bereiche nicht von normalen Kommentaren unterscheiden können. So fügt der Java Editor von Eclipse automatisch eine Import Anweisung in den Header einer Klasse ein, sobald eine Klasse eines anderen Packages verwendet wird. Die Anweisung wird nicht in den dafür vorgesehenen geschützten Bereich eingetragen und bei einer erneuten Generierung gelöscht. Hier muss ein Entwickler sorgfältig vorgehen und kann nicht alle Features eines Editorwerkzeugs nutzen.

Code-Qualität

Zumindest der b+m Generator liefert nur sehr beschränkte Qualität in der Formatierung des generierten Quellcodes und gewinnt damit keinen Blumentopf. Andere Kriterien wie z. B. mehrzeilige Deklaration und Instanziierung eines Objekts bestimmt man im Detail selbst durch die Formulierung der Templates. Der Einsatz eines Code-Formatter Werkzeugs ist unumgänglich, es sei denn, man ändert die Generator-Implementierung ab.

9.2 Bewertung

Zunächst lässt sich feststellen, dass mit dem Einsatz derzeitiger MDA konformer Werkzeuge wie etwa dem b+m Generator Framework ein kompletter Architekturrahmen aus einem abstrakten Modell generiert werden kann, dessen Code mit entsprechenden manuellen Ergänzungen auch ausführbar ist. Es konnte auch gezeigt werden, dass gegenüber herkömmlichen Generatoren eine Trennung zwischen Fachlichkeit und technischer Umsetzung erreicht werden kann. Ein Gewinn ist die abstrakte Dokumentation einer Architektur, die einfacher zu lesen ist als ein rein technisches Modell.
Man muss sich jedoch klar sein, dass zum momentanen Zeitpunkt nicht zwangsläufig mehr Code generiert wird als mit bisherigen Generatoren. Nicht zu unterschätzen ist die zusätzliche Komplexität, die durch Metamodelle und Template-Sprachen hinzukommt. Es muss wenigstens eine Person in einem Projekt geben, der/die sich damit auskennt.
Die Austauschbarkeit von Modellen über XMI ist eingeschränkt, bei der Wahl der Werkzeuge muss man sorgfältig bewerten, ob sie flexibel und zukunftsfähig sind, bevor man sich festlegt. Die Erstellung von Templates ist aufgrund mangelnder Editor-Unterstützung zeitauf-

wändig und fehleranfällig. Eine nachträgliche Einführung von MDA in Projekten mit bereits bestehenden Datenstrukturen und Modellen wie bei der Diplomarbeit geschehen, ist nicht unbedingt zu empfehlen, da neue Modelle (PIMs) gezeichnet werden und sich am Datenbankmodell orientieren müssen. Die mir bekannten Werkzeuge erlauben kein Round-Tripping und Debugging von Modellen, was für Entwickler ein wichtiges Kriterium für die Akzeptanz eines neuen Ansatzes darstellt.

10 Ausblick

MDA wurde erst vor kurzem „geboren“ und steht noch am Anfang einer weiteren Entwicklung. Die grundlegende Idee ist durchaus sinnvoll und durchdacht, man muss MDA jedoch noch eher etwas abwartend gegenüberstehen und den sinnvollen Einsatz je nach Projekt bewerten.

Solange existierende Werkzeuge nicht eine vollständig integrierte Unterstützung mit Roundtripping und Debugging Möglichkeiten auf Modellebene bieten, werden Entwickler sich nicht leicht davon überzeugen lassen und weiterhin herkömmliche Code-Generatoren verwenden, oder aus technischen Modellen Code generieren. Nicht jede Problemdomäne eignet sich für den Einsatz von MDA. Bisher wurde MDA hauptsächlich für die Generierung der technischen Infrastruktur in verteilten Systemen eingesetzt.

Die ersten pragmatischen Ansätze in Form von Generatoren, die eine Transformation von PIMs in Quellcode erlauben, zeigen durchaus brauchbare Resultate. Ob sich die modellierten PIMs tatsächlich über längere Zeit weiterverwenden lassen und synchron mit konkreten Implementierungen bleiben, ist noch nicht erwiesen.

Namhafte Koriphäen wie Martin Fowler und Dave Thomas [Tho01] sehen die Zukunft von MDA und UML eher skeptisch. Die von der OMG erwartete Revolution [OMG06] in der Softwareentwicklung wird laut [Fow01] nicht eintreten. Der momentane Hype um das Thema MDA erinnere zu stark an die Diskussion um CASE Tools in den 80er Jahren.

Die Diplomarbeit zeigt auch, welche Schwierigkeiten momentan noch mit der Umsetzung von MDA verbunden sind. Es wird deutlich, dass man am Anfang eines neuen Schritts in der Evolution der Disziplin der Software-Entwicklung ist und auch ein Umdenken bei den momentan praktizierenden Entwickern erforderlich ist, damit MDA auf längere Zeit einen Erfolg haben kann.

Trotz der momentan vorhanden Nachteile und Einschränkungen war es für mich erstaunlich, dass man aus einem relativ einfachen und abstrakten Modell einen komplexen Architekturrahmen generieren kann. Dies wird am besten deutlich durch den direkten optischen Vergleich der PIM Diagramme und der Diagramme des Architekturrahmens (vgl. Kapitel 8.1.4 und Kapitel 8.2.4). Man hat an dieser Stelle den bereits erwähnten Zugewinn an Dokumentation und Übersichtlichkeit.

Die Diplomarbeit lässt auch Raum offen für weitere Diplomarbeitsthemen, die sich mit der direkten Weiterführung des MDA Ansatzes mithilfe des b+m Generator Frameworks beschäftigen könnten. So wäre es interessant, auch das Frontend (beispielsweise basierend auf dem Struts-Framework) zu generieren oder Aktivitäts- bzw. Zustandsdiagramme mit den vorhandenen Klassendiagrammen zu verknüpfen.

A Listings

A.1 map.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ============================================================
This file maps class and attribute names and associations of
BusinessEntity classes in a PIM (UML Diagram) to their
corresponding database table and column names.

Author: Christian Märkle
Date: 17/02/04
============================================================== -->



<!--
The name of an attribute in a BusinessEntity class.
-->
<!ELEMENT cmp-field (#PCDATA)>

<!--
The name of the database.
-->
<!ELEMENT data-source-name (#PCDATA)>

<!--
The name of the database table column that corresponds to
an attribute in a BusinessEntity class.
-->
<!ELEMENT dbms-column (#PCDATA)>

<!--
This element has one or more entity elements that
represent the BusinessEntity classes in a PIM UML Diagram.
-->
<!ELEMENT entities (entity+)>

<!--
An <entity> element represents a BusinessEntity class. The
name and the attributes of this class will be mapped to
the corresponding database table and field names. Each
entity can be assigned to a data source.
-->
<!ELEMENT entity (name, data-source-name, table-name, field-map+)>

<!--
The <field-map> maps an attribute name of a BusinessEntity class to
```

```
a corresponding table column name in the database.
-->
<!ELEMENT field-map (cmp-field, dbms-column)>

<!--
The <foreign-key> maps a foreign key attribute name to a
foreign key table column name. Note, that a BusinessEntity class does
not represent foreign key fields as attributes with one exception:
The foreign key is a primary key at the same time.
The foreign key attribute names are needed by Value Objects that
represent database tables directly rather than Models or EntityBeans.
-->
<!ELEMENT foreign-key (cmp-field, dbms-column)>

<!--
The <key-map> assigns a <foreign-key> to a <primary-key>.
-->
<!ELEMENT key-map (foreign-key, primary-key)>

<!--
The root element that can hold a relations and an entities container.
-->
<!ELEMENT map (relations, entities)>

<!--
The logic name of a BusinessEntity
-->
<!ELEMENT name (#PCDATA)>

<!--
The <primary-key> maps a primary key attribute name to a
primary key table column name. Unlike a foreign key table column, a
primary key tabel column is represented as attribute in a BusinessEntity class.
-->
<!ELEMENT primary-key (cmp-field, dbms-column)>

<!--
A <relation> represents an association between two BusinessEntity classes.
The association on object level is identified by a
relation-role-name. The role-name is concatenated by the names of each
association end. On database level, a relation is identified by pairs of
foreign-key/primary-key assignments, represented by one or more <key-map>
elements here.
-->
<!ELEMENT relation (relation-role-name, key-map+)>

<!--
The unique name of an association/relation.
```

```
-->
<!ELEMENT relation-role-name (#PCDATA)>

<!--
A BusinessEntity class can have 0 or more associations to other BusinessEntities.
The <relations> element groups one or more <relation> elements.
-->
<!ELEMENT relations (relation+)>

<!--
The database table name of the corresponding BusinessEntity class.
-->
<!ELEMENT table-name (#PCDATA)>
```

Listing 16: map.dtd

A.2 config.properties (Aftersales Bonussystem)

```
# config.properties: Aftersales Bonussystem (VERA)
common.path=de/softlab/vera
BusinessEntity=server/be
BusinessActivity=server/ba
BusinessFacade=common/bf
common.exception.path=de/softlab/mda/common/exception
server.base.path=de/softlab/mda/server/base
server.util.path=de/softlab/mda/server/util
local=false
```

Listing 17: config.properties (Aftersales Bonussystem)

A.3 config.properties (Händler-Stammdatenverwaltung)

```
# config.properties: Haendler-Stammdatenverwaltung (IVS-R)
common.path=de/softlab/ivsr
BusinessEntity=server/be
BusinessActivity=server/ba
BusinessFacade=common/bf
common.exception.path=de/softlab/mda/common/exception
server.base.path=de/softlab/mda/server/base
server.util.path=de/softlab/mda/server/util
local=false
```

Listing 18: config.properties (Händler-Stammdatenverwaltung

B Konfigurationsdateien

A.4 config.properties

Die Datei config.properties dient zur Parametrisierung des Transformationsprozesses (Generierung) abhängig von projektspezifischen Einstellungen. Dies umfasst zum Stand der Diplomarbeit die Angabe von Pfaden/Packages für die generierten Artefakte, sowie die Möglichkeit, Local oder Remote EJB Komponenten zu erzeugen.
Auf config.properties wird aus den Templates über das erweiterte Metamodell durch die Klasse `de.softlab.mda.metamodel.PropertyProvider` zugegriffen. Die Datei muss sich im Verzeichnis /config unterhalb des Projektverzeichnisses befinden.

Parameter	Beschreibung
common.path	Der allgemeine Pfad-Anteil, endet üblicherweise mit dem Projektnamen, Bsp.: de/softlab/ivsr
BusinessEntity	Der Pfad-Anteil für die BusinessEntity-Klassen, Bsp.: server/be
BusinessActivity	Der Pfad-Anteil für die BusinessActivity-Klassen, Bsp.: server/ba
BusinessFacade	Der Pfad-Anteil für die BusinessFacade-Klassen, Bsp.: common/bf
common.exception.path	Der allgemeine Pfad, in den die Exception-Klassen platziert sind, Bsp.: de/softlab/mda/common/exception
server.base.path	Der Pfad der nicht generierten Basisklassen, Bsp.: de/softlab/mda/server/base
server.util.path	Der Pfad der nicht generierten Utility-Klassen, Bsp.: de/softlab/mda/server/util
local	Ein Flag, das bestimmt, ob EJB Komponenten Local oder Remote generiert werden sollen, mögliche Werte: true l false

Tabelle 3: Parameterbeschreibung config.properties

Um eine größere Flexibilität bei den Pfad- und Package-Angaben zu haben, werden diese aus mehreren Parametern zusammengesetzt, die Trennung der Verzeichnisse muss durch ein „/“ erfolgen.

common.path + BusinessEntity -> de/softlab/ivsr/server/be (de.softlab.ivsr.server.be)
common.path + BusinessActivity -> de/softlab/ivsr/server/ba (de.softlab.ivsr.server.ba)
usw.

Die Pfad-Angaben entsprechen dem Package-Namensraum. Die Pfade werden im Metamodell gleichzeitig als Package-Namen verwendet, der Slash „/“ wird dabei durch „.“ ersetzt.

A.5 map.xml

In map.xml werden Attribute, Klassennamen und Beziehungen des PIM den technischen Namen von Tabellen, Spalten und Primär-/Fremdschlüssel der Datenbank zugeordnet. Die

Datei wird von der Klasse `de.softlab.mda.metamodel.Mapper` eingelesen, die die instanziierten Metamodell-Klassen entsprechend konfiguriert. Die Syntax des XML-Dialekts orientiert sich an den EJB Deployment Deskriptoren (map.xml eine Art abgespeckter Deployment Deskriptor).

Element	Beschreibung
cmp-field	Der Name eines Attributs in einer BusinessEntity-Klasse
data-source-name	Der Name der Datenbank
dbms-column	Der Name einer Tabellenspalte auf der Datenbank
entities	Gruppiert eine oder mehrere <entity> Elemente
entity	Ein <entity> Element enspricht einer BusinessEntity-Klasse. Der Name und die Attribute der Klasse werden auf einen entsprechenden Tabellenname und Spaltennamen in der Datenbank abgebildet.
field-map	Ein <field-map> Element ordnet ein Attribut einer Tabellenspalte zu.
foreign-key	Ein <foreign-key> Element bildet den Namen eines Fremdschlüsselattributs auf den Namen einer Fremdschlüsselspalte ab. **Hinweis:** In einer BusinessEntity-Klasse gibt es keine Fremdschlüsselattribute, es sei denn, sie sind gleichzeitig Attribute des Primärschlüssels. Der Name wird trotzdem für die Value Objects benötigt, da diese anders als Models oder BusinessEntities **alle** Spalten der Datenbanktabellen direkt als Attribute abgebildet werden.
key-map	Ein <key-map> Element ordnet einen Primärschlüssel einem Fremdschlüssel zu.
map	Das Root Element der Datei.
name	Der logische Name einer BusinessEntity.
primary-key	Ein <primary-key> Element bildet den Namen eines Primärschlüsselattributs auf den Namen einer Primärschlüsselspalte ab.
relation	Ein <relation> Element entspricht einer Assozation zwischen zwei BusinessEntity-Klassen auf Objektebene und einer Primär-/Fremdschlüssel-Zuordnung auf Datenbankebene. Eine Assoziation besteht aus einem eindeutigen Rollennamen, der sich aus den Namen der beiden Assoziationsenden zusammensetzt.
relation-role-name	Der eindeutige Name einer Assoziation/Relation.
relations	Gruppiert mehrere <relation> Elemente.
table-name	Der Name der Datenbanktabelle, der einer BusinessEntity-Klasse entspricht

Tabelle 4: Beschreibung der Elemente von map.xml

C Literatur

Grundlagen

[Gam94] E. Gamma, R. Helm, R. Johnson, J. Vlissides
Design Patterns
Addison-Wesley, 1994

[Akc00] Celal Akcicek, TU Hamburg-Harburg
Objekttransformationen bei einem Paradigmenwechsel: Konzepte und Werkzeuge zur Abbildung von Programmobjekten auf Datenbankrelationen
Stand: 2000

Model Driven Architecture

[OMG01] Object Management Group
UML 2.0 Superstructure Specification v2.0
Stand: August 2003

[OMG02] Object Management Group
Meta Object Facility (MOF) Specification, v1.4
Stand: April 2002

[OMG03] Object Management Group
Model Driven Architecture (MDA), Architecture Board ORMSC1 Draft
Stand: 9.7.2001

[OMG04] Object Management Group
UML Profile for Enterprise Distributed Object Computing Specification
Stand: Februar 2002

[OMG05] Object Management Group
XML Metadata Specification Version 2.0
Stand: Mai 2003

[OMG06] Object Management Group
MDA Guide Version 1.0
Stand: Mai 2003

[b+m01] b+m Informatik AG
Generative Development Process
Stand: Januar 2003

[b+m02] b+m Informatik AG
b+m Generator FrameWork Referenz
Stand: Februar 2003

[Völ03] Markus Völter
Metamodellbasierte Codegenerierung
Java Spektrum Nr. 5 10/03

[Hub03] R. Hubert, A. Uhl
Quo vadis MDA?
ObjektSpektrum Nr. 1 02/04

[Boh03] M. Bohlen, Dr. G. Starke
MDA entzaubert
ObjektSpektrum Nr. 3 06/03

[Neu03] W. Neuhaus, C. Robitzki
Eine praktische Interpretation
Javamagazin 09/03

[Fow01] Martin Fowler
Model Driven Architecture
http://martinfowler.com/bliki/ModelDrivenArchitecture.html

[Tho01] Dave Thomas, Bederra Corp.
UML – Unified or Universal Language?
http://www.jot.fm/issues/issue_2003_01/column1

Enterprise JavaBeans

[Rom02] E. Roman, S. Ambler, T. Jewell
Mastering Enterprise Java Beans, Second Edition
John Wiley and Sons, Inc., 2002
http://www.theserverside.com

[Sha03] Bill Shannon
Java 2 Platform Enterprise Edition 1.4 Specification, Proposed Final Draft 3
Sun Microsystems

[Dem03] Linda G. DeMichiel
Enterprise JavaBeans 2.1 Specification, Final Release
Sun Microsystems

[Mar02] Floyd Marinescu
EJB Design Patterns
John Wiley & Sons, Inc., 2002
http://www.theserverside.com

Interne Dokumente

[Web02] Volker Weber, IBM
Component Architecture - Conceptual Architecture v1.0

[Raa01] Richard Raab, Softlab GmbH
Fachkonzept AS Bonussystem
Stand: 24.10.2003

[Ban01] Klaus Banzer, Softlab GmbH
System Design Master Data Administration BE

Stand: 16.02.2004

[Lau01] Nils Lausberg
System Proposal: Use Case Maintain GroupDealers
Stand: 29.10.2002

[Lau02] Nils Lausberg
System Proposal: Use Case Maintain DomesticDealers
Stand: 29.10.2002

D Externe Ressourcen

[ArcStyler] Interactive Objects Software
ArcStyler MDA Software Development Platform
http://www.io-software.de

[b+mGenFw] b+m Informatik AG
open b+m Generator FrameWork
http://www.architectureware.de
http://sourceforge.net/projects/architecturware

[Jalopy] Marco Hunsicker
Jalopy Code Formatter
http://jalopy.sourceforge.net/

[Ant] Apache Software Foundation
Apache Ant
http://ant.apache.org/

[Eclipse] Eclipse Foundation
Eclipse Universal Tool Platform
www.eclipse.org

[DB2] IBM
DB2 Database System
http://www-306.ibm.com/software/data/db2/

[Weblogic] BEA Systems, Inc.
BEA Weblogic Application Server Platform
www.bea.com

[AndroMDA] Matthias Bohlen
AndroMDA Code Generation Framework
http://sourceforge.net/projects/andromda

[JUnit] Erich Gamma, Kent Beck
JUnit Testing Framework
http://www.junit.org

[Poseidon] Gentleware AG
Poseidon for UML
www.gentleware.de

[VSS] Marcus Nylander
Visual Sourcesafe Plugin for Eclipse
http://vssplugin.sourceforge.net/

[Javadoc01] Christian Märkle
Javadoc Metamodell-Erweiterung
MDA-Generator\metamodel\doc\index.html

E Abbildungsverzeichnis

Hier sind die im Dokument verwendeten Abbildungen, Listings und Tabellen aufgelistet.

Abbildungen

Listings

Tabellen

CD-ROM zu dieser Diplomarbeit

Inhalt:

- die vorliegende Diplomarbeit im PDF-Format
- die Dokumentation der Metamodell-Erweiterung als Javadoc
- das MDA-Generator-Projekt mit der Metamodell-Erweiterung und den Templates
- die beiden Projektbeispiele „Händler-Stammdatenverwaltung“ und „Aftersales Bonus System“
- die in der Diplomarbeit angegebenen Online-Quellen